GRAPH AND GEOMETRIC ALGORITHMS ON DISTRIBUTED NETWORKS AND DATABASES

A Thesis
Presented to
The Academic Faculty

by

Danupon Nanongkai

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Algorithms, Combinatorics, and Optimization

Georgia Institute of Technology
August 2011

GRAPH AND GEOMETRIC ALGORITHMS ON DISTRIBUTED NETWORKS AND DATABASES

Approved by:

Professor Richard J. Lipton, Advisor
College of Computing
Georgia Institute of Technology

Professor H. Venkateswaran
College of Computing
Georgia Institute of Technology

Professor Gopal Pandurangan
Mathematical Sciences Division
Nanyang Technological University

Professor Jun Xu
College of Computing
Georgia Institute of Technology

Professor Prasad Tetali
College of Computing
Georgia Institute of Technology

Date Approved: 4 Mar 2011

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dick Lipton, for the support and freedom he gave throughout my stay at Georgia Tech. The discussions with him inspired me on many research topics and taught me the use of "crazy" ideas. His ability to explain complicated mathematical theories to the layman is one goal I will strive to achieve.

I am also very grateful to the ACO program under the lead of Robin Thomas. The financial support from the program was very important to many aspects of my research, from freeing me from TA work so that I can focus on research and supporting me to many conferences. It was also a great opportunity to take ACO courses and meet people from different departments.

I am deeply thankful to my constant collaborators and good friends Atish Das Sarma, Jittat Fakcharoenphol, Parinya Chalermsook, and Bundit Laekhanukit. They have always been my trustful sources of advice. Their contributions to my career is too much to describe here as this thesis will be doubled in length. I will miss endless discussions with Atish in front of whiteboards and poker tables, as well as those with Jittat, Parinya, and Bundit over beer.

I would like to thank Gopal Pandurangan and Ashwin Lall for their collaborations and advices. Gopal has supported me in many ways and anything I know about distributed computing is due to him. Ashwin has taught me many aspects of experimental research. I am also very grateful to have a chance to collaborate with Jim Xu, Prasad Tetali, Lap Chi Lau, Khaled Elbassioni, Kazuhisa Makino, and Julian Mestre.

I also thank members in the theory group at Georgia Tech for their support, especially Subrahmanyam Kalyanasundaram and Shiva Kintali. Subruk always provided

me helpful comments on my papers and presentations. Shiva is a great source of "inside" information. I also thank the Thai student community at Georgia Tech for making my stay here enjoyable.

Finally, and most importantly, I cannot thank enough my family for all their love and supports. My love of learning would not grow this tall without my father buying me all kinds of books and toys and bringing me to interesting places. Although I failed my mother as I had no interest in business, she always supported my decision in going into research in every way she could. My younger brother, Pat, is always available when I need hands. My girlfriend, Gib, is my great source of joy, strength, inspiration and encouragement. I cannot imagine how I could survive these hard years without them.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . **iii**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

SUMMARY . . . . . **xiii**

I INTRODUCTION **1**

1.1 Basic Models of Computation 2

1.1.1 Distributed Algorithms 2

1.1.2 Multi-pass Streaming Algorithms 6

1.1.3 Communication Complexity 9

1.2 Simple connections between models 10

1.3 How fast can we compute random walks on distributed networks? . . 13

1.3.1 Problems 14

1.3.2 Results 17

1.3.3 Related Work 20

1.4 How fast are approximation algorithms on distributed networks? . . 21

1.4.1 Problem 21

1.4.2 Results 24

1.5 How fast can we compute skylines on data streams and distributed networks? 27

1.5.1 Problem 27

1.5.2 Results 29

1.5.3 Related Work 30

1.6 How fast can we solve graph problems on data streams? 32

1.6.1 Problem 32

1.6.2 Results 37

1.7 Organization of this thesis 38

II RANDOM WALK ALGORITHMS AND APPLICATIONS . . . 39

2.1 Algorithm for 1-RW-DoS . . . 40

2.2 Analysis of Single-Random-Walk algorithm . . . 47

2.2.1 Correctness . . . 49

2.2.2 Analysis of Phase 1 . . . 51

2.2.3 Worst-case bound of Phase 2 . . . 52

2.2.4 A Probabilistic bound for Phase 2 . . . 53

2.3 Proof of Random Walk Visits Lemma (cf. Lemma 2.12) . . . 57

2.4 Variations, Extensions, and Generalizations . . . 60

2.4.1 Computing k Random Walks . . . 60

2.4.2 Regenerating the entire random walk . . . 61

2.4.3 Generalization to the Metropolis-Hastings algorithm . . . 61

2.5 Applications . . . 64

2.5.1 A Distributed Algorithm for Random Spanning Tree . . . 65

2.5.2 Decentralized Estimation of Mixing Time . . . 66

2.6 Conclusions . . . 69

III FROM COMMUNICATION COMPLEXITY TO DISTRIBUTED ALGORITHM LOWER BOUNDS . . . 71

3.1 Two-party distributed communication complexity . . . 71

3.2 Descriptions of $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$ . . . 75

3.2.1 Description of $G_1(\Gamma, p, d)$ . . . 75

3.2.2 Preliminary: the network $F(\Gamma, \kappa, \Lambda)$ . . . 76

3.2.3 Description of $G_2(\Gamma, \Lambda, \kappa)$ . . . 77

3.3 Proof of Theorem 3.3 . . . 79

3.3.1 Terminologies . . . 79

3.3.2 Proof . . . 82

3.4 Proof of Theorem 3.4 . . . 83

3.4.1 Terminologies . . . 83

3.4.2 Proof . . . 84

IV A TIGHT LOWER BOUND ON DISTRIBUTED RANDOM WALK COMPUTATION 91

4.1 The pointer chasing problem 92

4.2 Proof of Theorem 4.1 93

4.3 Conclusions 96

V DISTRIBUTED VERIFICATION AND HARDNESS OF DISTRIBUTED APPROXIMATION 98

5.1 Overview of Technical Approach 98

5.2 The Set-Disjointness problem 100

5.3 Randomized Lower Bounds for Distributed Verification 101

5.4 Hardness of Distributed Approximation 104

5.5 The rest randomized lower bounds 106

5.5.1 Randomized lower bound of s-t connectivity verification . . . 107

5.5.2 A randomized lower bound for cycle containment, e-cycle containment, and bipartiteness verification problem 109

5.5.3 Randomized lower bounds of connectivity, k-component, cut, s-t cut, least-element list, and edge on all paths verification . 111

5.6 One-sided Error Lower Bound of Spanning Tree Verification 113

5.6.1 A one-sided randomized lower bound of Hamiltonian cycle problem 113

5.6.2 Proof of Theorem 5.14 (sketched) 114

5.7 Details of hardness of approximation 118

5.8 Tightness of lower bounds 122

5.9 Conclusions 124

VI SKYLINE COMPUTATION ON MULTI-PASS STREAMS AND DISTRIBUTED NETWORKS 126

6.1 Streaming algorithms 127

6.1.1 Key idea: Eliminating points in a few passes 128

6.1.2 Streaming algorithm 132

6.1.3 Fixed-window algorithm 135

6.1.4 Algorithm Comparisons . . . 137
6.2 An almost tight lower bound of streaming skyline algorithms . . . 139
6.3 Extensions . . . 141
6.3.1 Deterministic 2D algorithm . . . 141
6.3.2 Posets . . . 143
6.4 Experimental Evaluation . . . 144
6.4.1 Baseline Comparisons . . . 145
6.4.2 Order Dependence . . . 149
6.5 Distributed algorithm . . . 151
6.6 Conclusions . . . 154
VII GRAPH ALGORITHMS ON BEST-ORDER STREAMS . . . 156
7.1 Models . . . 157
7.1.1 Best-Order Streaming Model . . . 157
7.1.2 Magic-Partition Communication Complexity . . . 159
7.1.3 Related models . . . 161
7.2 Detecting a Duplicate and Checking for Distinctness . . . 163
7.2.1 Space lower bound of the deterministic algorithms . . . 164
7.2.2 Randomized algorithm . . . 165
7.3 Perfect Matching . . . 166
7.3.1 Upper Bound . . . 166
7.3.2 Lower Bound . . . 167
7.4 Graph Connectivity . . . 171
7.4.1 Upper Bound . . . 171
7.4.2 Lower Bound . . . 174
7.5 Further Results . . . 175
7.5.1 Bipartite k-Regular graph . . . 175
7.5.2 Hamiltonian Cycle . . . 176
7.5.3 Non-Bipartiteness . . . 177

7.6 Conclusions 178

LIST OF TABLES

1 Lower bounds of randomized α-approximation algorithms on graphs of various diameters. Bounds in the first column are for the MST and shortest path tree problems [55] while those in the second column are for these problems and many problems listed in Fig. 11. We note that these bounds almost match the $O(\sqrt{n}\log^* n + D)$ upper bound for the MST problem [62, 96] and are independent of the approximation factor α. 26

2 Comparison of algorithms 138

LIST OF FIGURES

1 One way to view distributed algorithms in the $\mathcal{CONGEST}$ model is to view the computation as two-phase process, *days* and *nights*. In each day, each node sends one bit to each of its neighbors. (A node could send different bits to different neighbors.) In each night, nodes perform local computation. Since the local computation is free, we assume that any computation is finished in one night. Then again nodes exchange bits in the next days. The running time of the algorithm is measured by the number of days and nights required for the computation. . . . 5

2 Example of distributed algorithm for finding a spanning tree. 6

3 Multi-pass streaming algorithm reads the input one item at a time (each item in this case is a pair of numbers) from a huge source of data (e.g., harddisk) using a small memory (e.g., RAM) in a sequential manner. Once the last item is read, the algorithm can start a new pass by reading the first input item again. 8

4 Communication complexity of the equality problem. The trivial solution (Alice sends x to Bob) requires b bits of communication. This solution turns out to be the best deterministic algorithm. However, there exists a better randomized algorithm. 10

5 Search result from `hotels.com`. 28

6 Figure illustrating the Algorithm of stitching short walks together. . . 43

7 An example of $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$. The dashed edges (in red) have one copy while other edges have infinitely many copies. 74

8 An example of $F(\Gamma, \kappa, \Lambda)$ where $\Lambda = 2$ and $2 \leq \kappa < 3$. 76

9 Examples of i-right sets. 80

10 An example of round 11 in the proof of Theorem 3.4 (see detail in Example 3.9). 84

11 Problems and reductions between them to obtain randomized and one-sided error randomized lower bounds. For all problems, we obtain lower bounds as in Figure 1. 100

12 Example of H for the spanning connected subgraph problem (marked with thick red edges) when $x = 0...10$ and $y = 1...00$. 103

13 Example of H for s-t connectivity problem (marked with thick red edges) when $x = 0...10$ and $y = 1...00$. 108

14 Example of H for the cycle and e-cycle containment and bipartiteness verification problem when $x = 0...10$ and $y = 1...00$. 109

15 Example of H for the ST verification problem (marked with thick red edges) when the input graph G_A and G_B is as shown. 116

16 Example of the modification of the tree-part of G. Edges and nodes in blue are added to the tree. 117

17 Example of the reduction from the communication complexity of the Hamiltonian cycle problem to the distributed Hamiltonian cycle verification problem. Red edges form subgraph H. 117

18 Example from the proof of Lemma 6.1. 130

19 Comparison of Real Datasets 144

20 Varying w as a fraction of n for House 146

21 Varying w as a fraction of n for Anti-correlated 146

22 Varying the number of points n 147

23 Varying the number of dimensions d 148

24 Running time on various datasets 148

25 House, sorted by decreasing entropy. Varying w as a fraction of n, and varying d, n. 150

26 Variation in times for real and synthetic data sets (entropy sorted) . . 150

SUMMARY

This thesis studies the powers and limits of graph and geometric algorithms when we face with massive data, sometimes distributed on several machines. In this case, efficient algorithms are often required to use *sublinear* resources, such as time, memory, and communication. Three specific models of our interest are *distributed networks*, *data streams*, and *communication complexity*. In these models, we study lower and upper bounds of many problems motivated by applications in networking and database areas, as follows.

How fast can we compute random walks on distributed networks? Performing random walks on networks is a fundamental primitive that has found applications in many areas of computer science, including distributed computing. However, all previous distributed algorithms that compute a random walk sample of length ℓ as a subroutine always do so naively, i.e., in $O(\ell)$ rounds. In this thesis, we show that a faster algorithm exists. We also show that our algorithm is optimal. Moreover, we show extensions and applications of our algorithm.

A sublinear-time algorithm. We present a distributed algorithm for performing random walks whose time complexity is sublinear in the length of the walk. Our algorithm performs a random walk of length ℓ in $\tilde{O}(\sqrt{\ell D})$ rounds ($\tilde{O}$ hides polylog n factors where n is the number of nodes in the network) on an undirected network, where D is the diameter of the network. For small diameter graphs, this is a significant improvement over the naive $O(\ell)$ bound.

A tight lower bound. We also show a tight unconditional lower bound for computing a random walk. Specifically, we show that for any n, D, and $D \leq \ell \leq (n/(D^3 \log n))^{1/4}$, performing a random walk of length $\Theta(\ell)$ on an n-node network of

diameter D requires $\Omega(\sqrt{\ell D} + D)$ time. This bound is *unconditional*, i.e., it holds for any (possibly randomized) algorithm. To the best of our knowledge, this is the first lower bound that the diameter plays a role of multiplicative factor. Our bound shows that the algorithm we developed is time optimal. Besides, our technique establishes a new connection between communication complexity and distributed algorithm lower bounds, leading to improved lower bounds of many problems, as we show later in this thesis.

Extensions and applications of random walk computation. We also extend our algorithms to efficiently perform k independent random walks in $\tilde{O}(\sqrt{k\ell D}+k)$ rounds. We also show that our algorithm can be applied to speedup the more general Metropolis-Hastings sampling. Our random walk algorithms can be used to speed up distributed algorithms in a variety of applications that use random walks as a subroutine. In particular, we present two main applications. First, we give a fast distributed algorithm for computing a random spanning tree (RST) in an arbitrary (undirected) network which runs in $\tilde{O}(\sqrt{m}D)$ rounds with high probability (m is the number of edges). Our second application is a fast decentralized algorithm for estimating mixing time and related parameters of the underlying network.

How fast are approximation algorithms on distributed networks? We show lower bounds of many approximation graph algorithms by studying the *verification* problem, stated as follows. Let H be a subgraph of a network G where each vertex of G knows which edges incident on it are in H. We would like to verify whether H has some properties, e.g., if it is a tree or if it is connected (every node knows in the end of the process whether H has the specified property or not). We would like to perform this verification in a decentralized fashion via a distributed algorithm. The time complexity of verification is measured as the number of rounds of distributed communication.

In this thesis we initiate a systematic study of distributed verification, and give almost tight lower bounds on the running time of distributed verification algorithms for many fundamental problems such as connectivity, spanning connected subgraph, and $s - t$ cut verification.

We then use this these results to derive strong unconditional time lower bounds on the *hardness of distributed approximation* for many classical optimization problems including minimum spanning tree, shortest paths, and minimum cut. Many of these results are the first non-trivial lower bounds for both exact and approximate distributed computation and they resolve previous open questions. Moreover, our unconditional lower bound of approximating minimum spanning tree (MST) subsumes and improves upon the previous hardness of approximation bound of Elkin [STOC 2004] as well as the lower bound for (exact) MST computation of Peleg and Rubinovich [FOCS 1999]. Our result implies that there can be no distributed approximation algorithm for MST that is significantly faster than the current exact algorithm, for *any* approximation factor.

As in the case of the lower bound of random walk computation, our lower bound proofs use the novel connection between communication complexity and distributed computing.

How fast can we compute skylines on data streams and distributed networks? The skyline query is a basic database operation that outputs maximal points in a set of multi-dimensional points. Efficiently processing this query has been studied in the database community for almost a decade. In this setting we need external algorithms since the database is usually stored on a disk. Although there are many previous algorithms that were analyzed in terms of I/Os, no formal models of external algorithms were studied before. In particular, the fact that sequential access is much faster than random access had not been exploited.

In this thesis, we study the skyline problem on the multi-pass streaming model. This model is the most restricted (least powerful) among many models that differentiates between sequential and random accesses as algorithms on this model are allowed to do a random access only once after each round of sequential accesses to the whole input.

We show that, even in this very restricted model, one can still get an efficient algorithm. Our algorithm uses space only enough to store the skyline and uses linear sequential and logarithmic random accesses (or passes in the terminology of data streams). To the best of our knowledge, it is the first randomized skyline algorithm in the literature. We also prove that this our algorithm is near-optimal.

Additionally, we show that the same idea can be used to develop a distributed algorithm that is near optimal. We also show that the algorithm can handle partially ordered domains on each attribute. Finally, we demonstrate the robustness of this algorithm via extensive experiments on both real and synthetic datasets. Our algorithm is comparable to the existing algorithms in average case and additionally tolerant to simple modifications of the data, while other algorithms degrade considerably with such variations.

How fast can we solve graph problems on data streams? Data streams are known to require large space to solve graph problems when data is presented in an adversarial order. However, assume an adversarial order is too pessimistic in practice. What if the data is in a random or sorted order, or pre-arranged in a suitable way? We explore the other end of the streaming model where the data is presented in the best order possible. We call this mode the *best-order streaming* model.

Roughly, this model is a proof system where a space-limited verifier has to verify a proof sequentially (i.e., it reads the proof as a stream). Moreover, the proof itself is just a specific ordering of the input data. This model is closely related to many

other models of computation such as communication complexity and proof checking, and could be used in applications such as cloud computing.

We focus on graph problems where the input is a sequence of edges. We show that even under this model, checking some basic graph properties deterministically requires linear space in the number of nodes. To contrast this, we show that randomized verifiers are powerful enough to check many graph properties in poly-logarithmic space. Therefore, there is still some hope for a suitable streaming model which should sit somewhere between the two extreme cases, and the randomness will be a crucial tool.

CHAPTER I

INTRODUCTION

In this age of internet and online social communities, our increasing ability to generate large amount of information forces many applications to deal with massive data set, often with the size of terabytes (and sometimes petabytes!) distributed on several machines. For this huge amount of data, polynomially computable is not considered efficient enough anymore. In this thesis, we study the powers and limits of fundamental graph and geometric algorithms when there are limited computational resources, such as time, memory, and communication, which are usually required to be sublinear in the input size. We focus on problems arose from networking and database areas. In particular, we are interested in the following problems.

- How fast can we compute random walks on distributed networks?
- How fast are approximation algorithms on distributed networks?
- How fast can we compute skylines on data streams and distributed networks?
- How fast can we solve graph problems on data streams?

To attack these problems, we need to study several models of computation. The models of our main interest are the variations of

- distributed networks,
- data streams, and
- communication complexity.

The first two models are useful in capturing scenarios where different computational resources are limited while the last model is useful in proving the limits of the first two models.

This chapter provide a basic knowledge required for the rest of this thesis. Section 1.1 introduces the basic models of distributed networks, data streams and communication complexity. Later in this thesis, we will see many variations of these models. Section 1.2 shows some simple reductions between problems on these models to give an idea how the models are related. More sophisticated reductions will appear later in the thesis. After we have describe the models, we are ready to describe the problems of our interest and our results. This is done from Section 1.3 to Section 1.6. The organization of the rest of this thesis can be found in Section 1.7

1.1 Basic Models of Computation

Our models of interest are distributed networks, data streams, communication complexity, and their variations. The first two models are useful in capturing scenarios where different computational resources are limited while the last model is used to prove the limits of the first two models. We now describe the basic versions of these models of computation before explaining our problems and results. Later in this thesis, we will introduce several variations of these models, as needed by different applications.

1.1.1 Distributed Algorithms

Large and complex networks, such as the human society, the Internet, or the brain, are being studied intensely by different branches of science. Each individual node in such a network can directly communicate only with its neighboring nodes. Despite being restricted to such *local* communication, the network itself should work towards a *global* goal, i.e., it should organize itself, or deliver a service.

We are mainly interested in the possibilities and limitations of distributed computation, i.e., to what degree local information is sufficient to solve global tasks. Many tasks can be solved entirely via local communication, for instance, how many friends of friends one has. Research in the last 30 years has shown that some classic combinatorial optimization problems such as matching, coloring, dominating set, or approximations thereof can be solved using small (i.e., polylogarithmic) local communication. For example, a maximal independent set can be computed in time $O(\log n)$ [108], but not in time $\Omega(\sqrt{\log n / \log\log n})$ [93] (n is the network size). This lower bound even holds if message sizes are unbounded.

However many important optimization problems are "global" problems from the distributed computation point of view. To count the total number of nodes, to determining the diameter of the system, or to compute a spanning tree, information necessarily must travel to the farthest nodes in a system. If exchanging a message over a single edge costs one time unit, one needs $\Omega(D)$ time units to compute the result, where D is the network diameter. If message size was unbounded, one can simply collect all the information in $O(D)$ time, and then compute the result. Hence, in order to arrive at a realistic problem, we need to introduce communication limits, i.e., each node can exchange messages with each of its neighbors in each step of a synchronous system, but each message can have at most B bits (typically B is small, say $O(\log n)$).

Specifically, consider an undirected, unweighted, connected n-node multi-graph $G = (V, E)$. Suppose that every node (vertex) hosts a processor with unbounded computational power, but with limited initial knowledge. Specifically, assume that each node is associated with a distinct identity number from the set $\{1, 2, ..., n\}$. At the beginning of the computation, each node v accepts as input its own identity number and the identity numbers of its neighbors in G. The node may also accept some additional inputs as specified by the problem at hand. The nodes are allowed to

communicate through the edges of the graph G. The communication is synchronous, and occurs in discrete pulses, called *rounds*. In particular, all the nodes wake up simultaneously at the beginning of round 1, and from this point on the nodes always know the number of the current round. In each round each node v is allowed to send an arbitrary message of size B through each edge $e = (v, u)$ that is adjacent to v, where B is the parameter of the network. The message will arrive to u at the end of the current round. This is a standard model of distributed computation known as the $\mathcal{CONGEST}(B)$ model or simply the B-model [128]. In many applications, we consider the model called $\mathcal{CONGEST}$ where we assume that $B = O(\log n)$. This model has been attracting a lot of research attention during last two decades (e.g., see [128] and the references therein).

There are several measures of efficiency of distributed algorithms, but we will concentrate on one of them, specifically, *the running time*, that is, the number of rounds of distributed communication. (Note that the computation that is performed by the nodes locally is free, i.e., it does not affect the number of rounds.) Many fundamental network problems such as minimum spanning tree, shortest paths, etc. have been addressed in this model (e.g., see [110, 128, 122]). In particular, there has been much research into designing very fast distributed approximation algorithms (that are even faster at the cost of producing sub-optimal solutions) for many of these problems (see e.g., [52, 51, 87, 86]). Such algorithms can be useful for large-scale resource-constrained and dynamic networks where running time is crucial.

Algorithms in this model can be viewed as having two phases, "days" and "nights", where one bit is communicated between neighbors in each day and local computation are done at night (we assume that any local computation can be finished in one night), as in Figure 1.

Example 1.1. Consider the problem of finding a spanning tree on a distributed network. This problem can be solved in $O(D)$ rounds as follows. First, we pick any

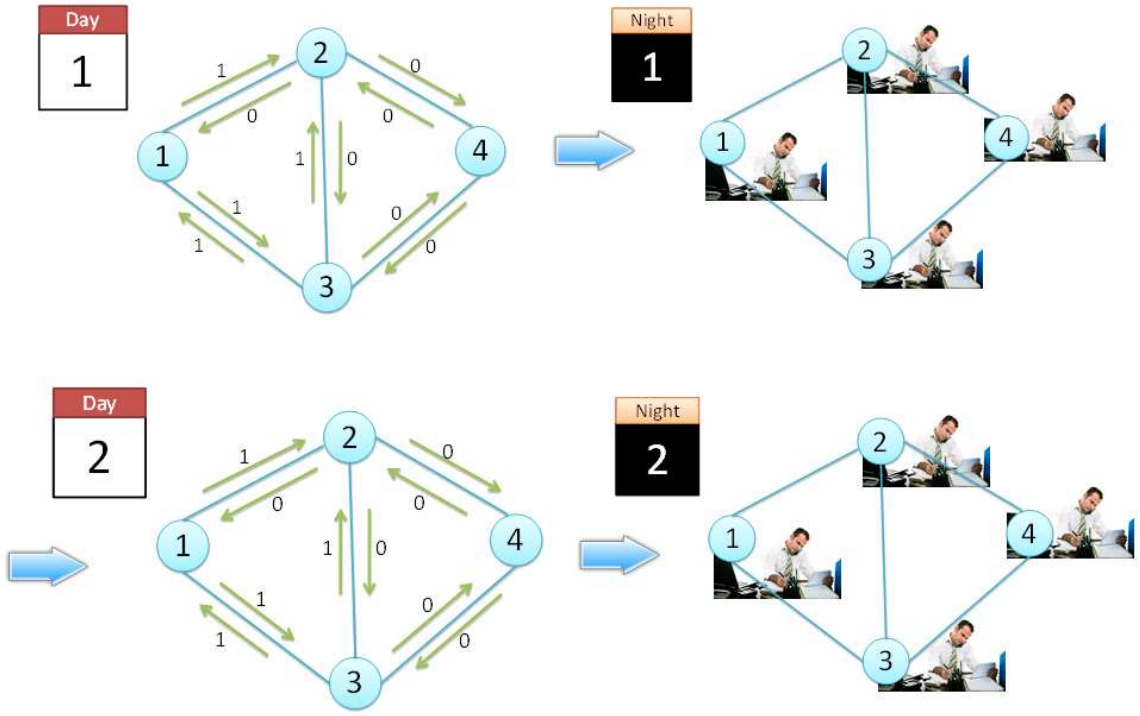


Figure 1: One way to view distributed algorithms in the $\mathcal{CONGEST}$ model is to view the computation as two-phase process, *days* and *nights*. In each day, each node sends one bit to each of its neighbors. (A node could send different bits to different neighbors.) In each night, nodes perform local computation. Since the local computation is free, we assume that any computation is finished in one night. Then again nodes exchange bits in the next days. The running time of the algorithm is measured by the number of days and nights required for the computation.

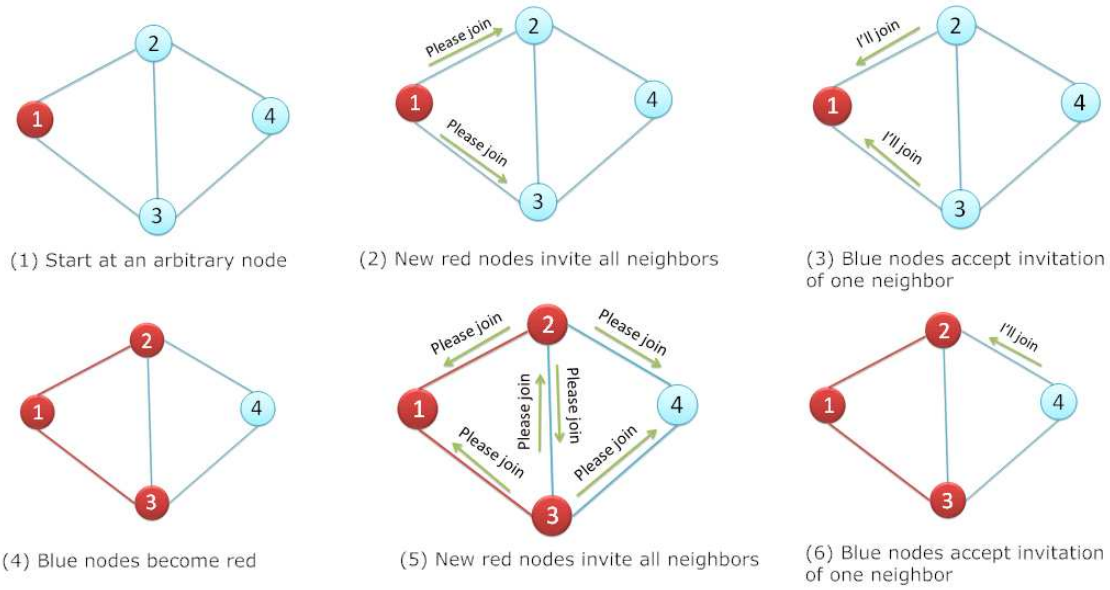


Figure 2: Example of distributed algorithm for finding a spanning tree.

node as a starting node (e.g., the node with the smallest ID). Call this node "red". (See Figure 2 (1) as an example.) In each round, new red nodes send invitations to all its neighbors (see, e.g., Figure 2 (2) and (5)). In the next rounds, nodes that are not red (say "blue") that receive invitations respond to one of the invitations from their neighbors (see, e.g., Figure 2 (3) and (6)). These nodes connect to the red nodes that they responded, thus forming a bigger spanning tree, and become new red nodes. The process repeats until all nodes are red. Observe that nodes of distance d from the starting node will become red after $O(d)$ rounds. Thus, all nodes will become red after $O(D)$ rounds.

1.1.2 Multi-pass Streaming Algorithms

Since most data nowadays are too large to fit in the main (internal) memory (e.g., RAM), they are typically stored in the external memory on one or more magnetic disks. In this type of memory, the sequential disk access is preferable to the random disk access for several reasons. First, the sequential disk access is considerably faster

than the random disk access as the latter involves a number of seek operations. For example, database algorithms for spatial join that access pre-existing index structures (and thus do random I/O) can often be slower in practice than algorithms that access substantially more data but in a sequential order (as in streaming) [9]. Second, sequential access has the advantage of using modern caching architectures optimally, making the algorithm independent of the block size (i.e., *cache-oblivious*) [60]. For these reasons, the models designed to capture magnetic disks have to distinguish the two types of memory access.

There have been many practical models proposed in the literature. Well known models include the *parallel disk model* (PDM) and the *multi-pass streaming model* [146, 132, 8, 76, 117]. In this thesis, we aim at exploring the power and limitation of the latter model in the context of graph and geometric algorithms.

In this multi-pass streaming model, the input stream $a_1, a_2, ..., a_m$ arrives sequentially, item by item. The streaming algorithm with limited memory space has to compute some function $f(a_1, a_2, ..., a_m)$ after it reads the whole stream. The point is that once an item a_i is read, the algorithm cannot go back to read a_{i-1} again. In some cases, the algorithm is allowed to read the stream in many *passes*; after it reads a_m, it may begin a new pass by reading the stream starting from a_1. In this case, the number of passes should be minimized. See Figure 3.

Example 1.2. Given n numbers $x_1, x_2, \ldots, x_n$, where $x_i \in \{1, 2, \ldots, n\}$, can we check whether these numbers are all distinct, using $O(\log n)$ bits of memory? In the RAM model where random access is allowed, we can do this by sorting the numbers and checking if all pair of consecutive numbers are distinct. However, when we are restricted to access these numbers in a streaming manner, sorting operation is expensive to perform. In fact, it can be shown that *any* deterministic algorithm requires $\Omega(n)$ bits of memory to solve this problem on the streaming model. Fortunately, there is a randomized $O(\log n)$-space algorithm that solves this problem. This is done by the

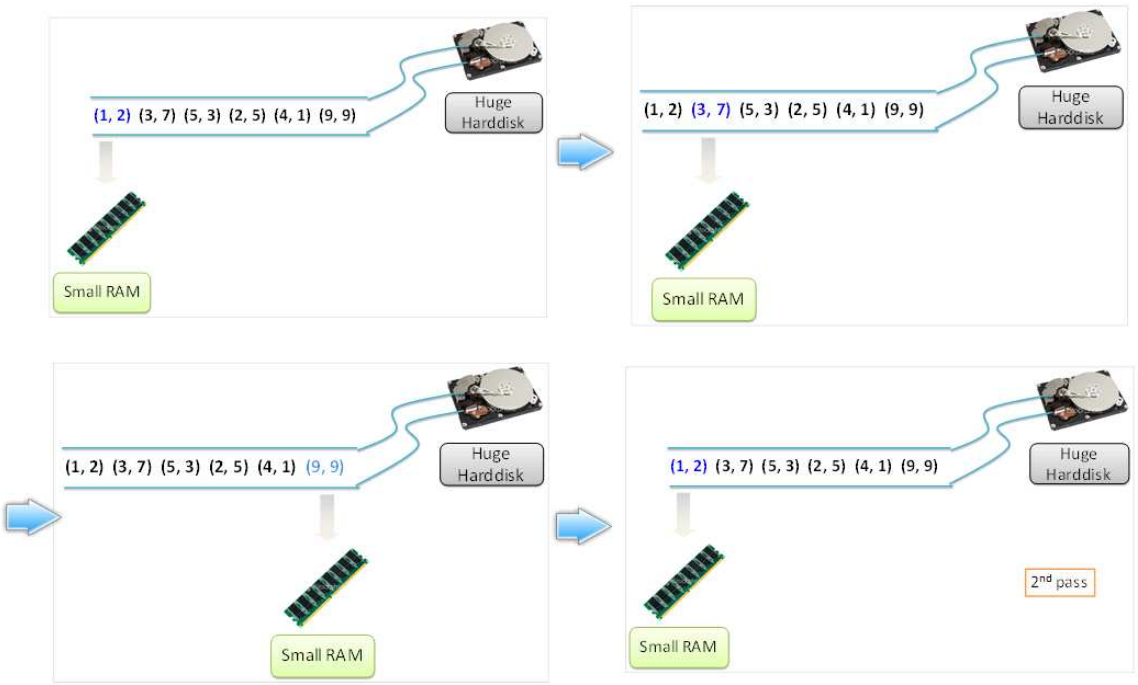


Figure 3: Multi-pass streaming algorithm reads the input one item at a time (each item in this case is a pair of numbers) from a huge source of data (e.g., harddisk) using a small memory (e.g., RAM) in a sequential manner. Once the last item is read, the algorithm can start a new pass by reading the first input item again.

fingerprinting technique where one computes $\Pi_{i=1}^{k}(x_i + r) \mod p$, for some random prime p and $r \in \{0, 1, \ldots, p-1\}$. We discuss this problem in greater details (and show both upper and lower bounds) in Chapter 7.

1.1.3 Communication Complexity

The basic model of communication complexity was introduced by Yao [152] and has found many applications such as circuit lower bounds, data structure lower bounds, streaming algorithms lower bounds, and distributed algorithm lower bounds. In this model, there are two parties that have unbounded computational power. Each party receives a b-bit string, for some integer $b \geq 1$, denoted by x and y in $\{0,1\}^b$. We call the party receiving x *Alice*, and the other party *Bob*. At the end of the process, Bob will output $f(x, y)$. They both want to together compute $f(x, y)$ for some function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$. To do this, they have to communicate with each other via a bidirectional edge of unlimited bandwidth. The goal is to minimize the number of bits communicated between them.

We consider the *public coin randomized algorithms* under this model. In particular, we assume that both parties share a random bit string of infinite length. For any $\epsilon \geq 0$, we say that a randomized algorithm $\mathcal{A}$ is *ϵ-error* if for any input, it outputs the correct answer with probability at least $1 - \epsilon$, where the probability is over all possible random bit strings. The *communication complexity* of $\mathcal{A}$ is the number of communication bits in the worst case (over all inputs and random strings). Let $R_{\epsilon}^{cc-pub}(f)$ denote the communication complexity of best ϵ-error algorithms.

In this thesis, we are interested in lower bounds of $R_{\epsilon}^{cc-pub}(f)$ under some functions f, as we will use them to show limits of distributed and streaming algorithms.

Example 1.3. Consider the following *equality* problem where Alice and Bob, upon receiving $x, y \in \{0,1\}^b$, want to check whether $x = y$. (See Figure 4.) That is, they want to compute a function EQ where $\text{EQ}(x, y)$ is one if $x = y$ and zero otherwise.

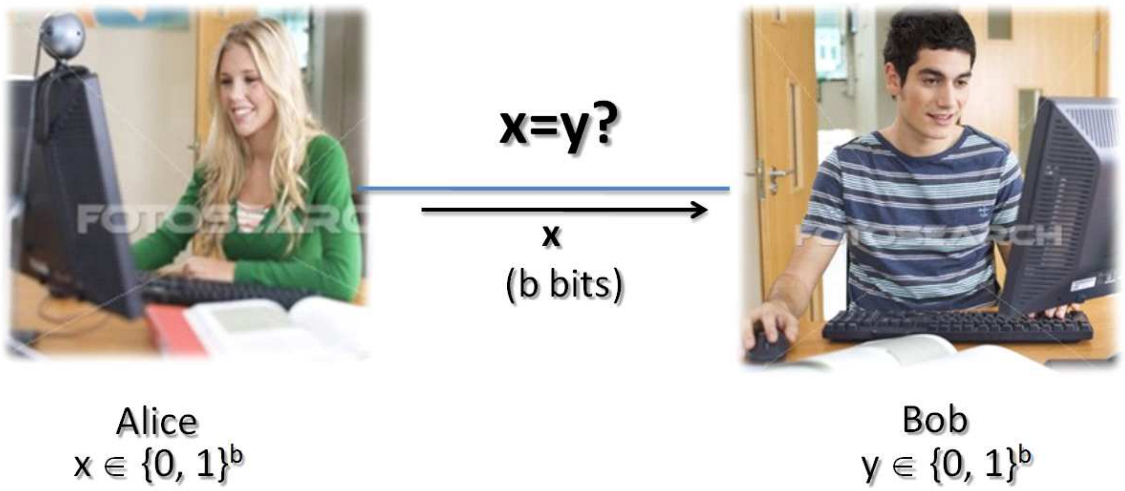


Figure 4: Communication complexity of the equality problem. The trivial solution (Alice sends x to Bob) requires b bits of communication. This solution turns out to be the best deterministic algorithm. However, there exists a better randomized algorithm.

A naive solution is having Alice send x to Bob. The communication complexity of this algorithm is $O(b)$ since Alice needs to send b bits to Bob. This solution turns out to be the best deterministic algorithm. In other words, one can show that $R_0^{cc-pub}(\text{EQ}) = \Omega(b)$. However, a randomized algorithm can do much better as one can using the fingerprint technique (described in Example 1.2) to solve the problem using $O(\log b)$ communication complexity.

1.2 Simple connections between models

In this thesis, we explore the interactions between the three aforementioned models and their variations, leading to improved lower and upper bounds of many problems. Before we do this, we show some simple connections between these models that are previously observed. Later in this thesis, we use some of these connections as tools to prove lower bounds and sometimes introduce new connections that help us break through previous barriers.

Recall that for any function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$, we let $R_\epsilon^{cc-pub}(f)$ denote

the communication complexity of the best ϵ-error algorithms for computing f.

From communication complexity lower bounds to pass-space tradeoff lower bounds of streaming algorithms. Consider computing a function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$ where its input is presented as a stream, i.e., for any input strings $x, y \in \{0,1\}^b$ we receive a stream $x_1, x_2, \ldots, x_b, y_1, y_2, \ldots, y_b$, where x_i (y_i, respectively) is the i^{th} bit of x (y, respectively), and we want to compute $f(x,y)$ using the least space and smallest number of passes. The following observation has been used to prove many lower bounds of streaming algorithms (see, e.g., [76, 8, 118]).

Observation 1.4. For any s, any ϵ-error streaming algorithm $\mathcal{A}$ that uses at most p passes requires at least $R_\epsilon^{cc-pub}(f)/(2p-1)$ bits of space to compute f.

Proof. Let $\mathcal{A}$ be an algorithm that uses at most p passes and s bits of space. Observe that Alice and Bob can use $\mathcal{A}$ to compute f as follows. First, Alice writes down her input $x_1, \ldots, x_b$, as if it is the first half of the input stream, and simulates $\mathcal{A}$ on such stream. Once she finishes simulating $\mathcal{A}$ on her input, she sends the data stored in the working space of $\mathcal{A}$ to Bob. This needs s bits of communication since $\mathcal{A}$ stores at most s bits in its memory. Then, Bob continue simulating $\mathcal{A}$ on $y_1, \ldots, y_b$. Once he is done, he again sends the data stored in the working space of $\mathcal{A}$ to Alice. If $\mathcal{A}$ needs p passes to compute f, then Alice and Bob have to send information to each other at most $2p-1$ times (Bob does not have to send a message to Alice in the last pass). Thus, they will exchange at most $s(2p-1)$ bits in total. This implies that $s(2p-1) \geq R_\epsilon^{cc-pub}(f)$. □

Later in this thesis, we will see a slightly different connection which connects between a variation of the communication complexity model, called *bounded-round* communication complexity, and the multi-pass streaming model. We use this to show a lower bound of computing skyline on multi-pass stream. (See Chapter 6.) We

also introduce another variation of communication complexity, called *magic-partition* communication complexity. We use this model to prove lower bounds of many graph problems on a variation of multi-pass stream called *best-order* stream (introduced in Section 1.6 and studied in Chapter 7).

From communication complexity lower bounds to distributed algorithm lower bounds. Consider computing a function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$ where its input strings x and y are given to any two nodes, denoted by s and t, in a B-model distributed network G, and we want to compute $f(x,y)$ using the smallest number of rounds. Recall that, in each round, each node can send at most B bits to its neighbors.

One simple method to prove lower bounds of this problem is to identify communication bottlenecks in the network, as follows. For any partition of the nodes in G into two sets S_1 and S_2, where $s \in S_1$ and $t \in S_2$, we let $P(S_1, S_2)$ be the number of edges connecting S_1 and S_2. The generalization of the following observation is used in proving lower bounds of some distributed algorithms (see, e.g., [95, 99]).

Observation 1.5. For any node partition (S_1,S_2), where $s \in S_1$ and $t \in S_2$, any ϵ-error distributed algorithm $\mathcal{A}$ for computing f requires at least $R_\epsilon^{cc-pub}(f)/(2BP(S_1, S_2))$ time.

Proof. Let $\mathcal{A}$ be an algorithm that uses at most t rounds of communication. Observe that Alice and Bob can use $\mathcal{A}$ to compute f as follows. Alice and Bob simulates the computation on nodes in S_1 and S_2, respectively. In each round of communication of $\mathcal{A}$, nodes send some messages to their neighbors. To be able to continue simulating computation on nodes in S_1, Alice has to know all messages sent to all nodes in S_1 in each round. To achieve this, Bob sends all messages sent from nodes in S_2 to nodes in S_1 in each round. Similarly, Bob can continue simulating computation on nodes in S_2 if Alice sends him all messages sent from nodes in S_2 to nodes in S_1 in each round.

One party sends at most $tB \cdot P(S_1, S_2)$ bits to the other party since, in each round, there are B bits sent on each of $P(S_1, S_2)$ edges from one set of nodes to the other. Therefore, $(2BP(S_1, S_2)) \cdot t \geq R_{\epsilon}^{cc-pub}(f)$. The observation follows. □

Later in this thesis, we develop a completely different connections between communication complexity and distributed algorithm. In particular, we consider distributed algorithms on a class of graphs that do not have a small bottleneck. The observation above is not useful for this class of graphs. We use the new connection to prove tight lower bounds of random walk computation and many approximation algorithms. (See Chapter 3, 4, and 5.)

1.3 How fast can we compute random walks on distributed networks?

Random walks play a central role in computer science, spanning a wide range of areas in both theory and practice. The focus of this thesis is on random walks on networks, in particular, decentralized algorithms for performing random walks in arbitrary networks. Random walks are used as an integral subroutine in a wide variety of network applications ranging from token management [77, 19, 37], load balancing [82], small-world routing [88], search [155, 2, 35, 64, 109], information propagation and gathering [20, 85], network topology construction [64, 98, 106], checking expander [50], constructing random spanning trees [22, 15, 12], monitoring overlays [115], group communication in ad-hoc network [49], gathering and dissemination of information over a network [6], distributed construction of expander networks [98], and peer-to-peer membership management [61, 156]. Random walks are also very useful in providing uniform and efficient solutions to distributed control of dynamic networks [23, 155]. Random walks are local and lightweight and require little index or state maintenance which make them especially attractive to self-organizing dynamic networks such as Internet overlay and ad hoc wireless networks.

A key purpose of random walks in many of these network applications is to perform node sampling. While the sampling requirements in different applications vary, whenever a true sample is required from a random walk of certain steps, typically all applications perform the walk naively — by simply passing a token from one node to its neighbor: thus to perform a random walk of length ℓ takes time linear in ℓ.

In this thesis, we present a sublinear time (sublinear in ℓ) distributed random walk sampling algorithm that is significantly faster than the naive algorithm when $\ell >> D$. Our algorithm runs in time $\tilde{O}(\sqrt{\ell D})$ rounds. Moreover, we show that this running time is optimal by showing a matching unconditional lower bound which is obtained by establishing a new connection between communication complexity and distributed algorithms.

We also present two key applications of our algorithm. The first is a fast distributed algorithm for computing a random spanning tree, a fundamental problem that has been studied widely in the classical setting (see e.g., [83] and references therein) and in some special cases in distributed settings [15]. To the best of our knowledge, our algorithm gives the fastest known running time in an arbitrary network. The second is to devising efficient decentralized algorithms for computing key global metrics of the underlying network — mixing time, spectral gap, and conductance. Such algorithms can be useful building blocks in the design of *topologically (self-)aware* networks, i.e., networks that can monitor and regulate themselves in a decentralized fashion. For example, efficiently computing the mixing time or the spectral gap, allows the network to monitor connectivity and expansion properties of the network.

1.3.1 Problems

We consider the following basic random walk problem.

Computing One Random Walk where Destination outputs Source (1-RW-DoS). We are given an arbitrary undirected, unweighted, and connected n–node network $G = (V, E)$ in the $\mathcal{CONGEST}$ model (defined in Section 1.1.1) and a source node $s \in V$. The goal is to devise a distributed algorithm such that, in the end, some node v outputs the ID of s, where v is a destination node picked according to the probability that it is the destination of a random walk of length ℓ starting at s. For short, this problem will henceforth be simply called *Single Random Walk*.

For clarity, observe that the following naive algorithm solves the above problem in $O(\ell)$ rounds: The walk of length ℓ is performed by sending a token for ℓ steps, picking a random neighbor in each step. Then, the destination node v of this walk outputs the ID of s. Our goal is to perform such sampling with significantly less number of rounds, i.e., in time that is sublinear in ℓ. On the other hand, we note that it can take too much time (as much as $\Theta(|E| + D)$ time) in the $\mathcal{CONGEST}$ model to collect all the topological information at the source node (and then computing the walk locally).

We also consider the following variations and generalizations of the Single Random Walk problem.

1. *k Random Walks, Destinations output Sources (k-RW-DoS)*: We have k sources $s_1, s_2, ..., s_k$ (not necessarily distinct) and we want each of k destinations to output an ID of its corresponding source.

2. *k Random Walks, Sources output Destinations (k-RW-SoD)*: Same as above but we want each source to output the ID of its corresponding destination.

3. *k Random Walks, Nodes know their Positions (k-RW-pos)*: Instead of outputting the ID of source or destination, we want each node to know its position(s) in the random walk. That is, for each s_i, if $v_1, v_2, ..., v_\ell$ (where $v_1 = s_i$) is the result random walk starting at s_i, we want each node v_j in the walk to

know a pair (s_i, j) at the end of the process. The pair (s_i, j) represents the fact that the node containing it is the j^{th} node in the random walk starting at s_i.

Motivation. There are two key motivations for obtaining sublinear time bounds. The first is that in many algorithmic applications, walks of length significantly greater than the network diameter are needed. For example, this is necessary in both the applications presented later in this thesis, namely distributed computation of a random spanning tree (RST) and computation of mixing time. In the RST algorithm, we need to perform a random walk of expected length $O(mD)$ (where m is the number of edges in the network). In decentralized computation of mixing time, we need to perform walks of length at least the mixing time which can be significantly larger than the diameter (e.g., in a random geometric graph model [119], a popular model for ad hoc networks, the mixing time can be larger than the diameter by a factor of $\Omega(\sqrt{n})$.) More generally, many real-world communication networks (e.g., ad hoc networks and peer-to-peer networks) have relatively small diameter, and random walks of length at least the diameter are usually performed for many sampling applications, i.e., $\ell >> D$. It should be noted that if the network is rapidly mixing/expanding which is sometimes the case in practice, then sampling from walks of length $\ell >> D$ is close to sampling from the steady state (degree) distribution; this can be done in $O(D)$ rounds (note however, that this gives only an approximately close sample, not the exact sample for that length). However, such an approach fails when ℓ is smaller than the mixing time.

The second motivation is understanding the time complexity of distributed random walks. Random walk is essentially a global problem which requires the algorithm to "traverse" the entire network. Classical "global" problems include the minimum spanning tree, shortest path etc. Network diameter is an inherent lower bound for such problems. Problems of this type raise the basic question whether n (or ℓ as the

case here) time is essential or is the network diameter D, the inherent parameter. As pointed out in the seminal work of [62], in the latter case, it would be desirable to design algorithms that have a better complexity for graphs with low diameter.

1.3.2 Results

An Optimal Sublinear-time Distributed Random Walk Computation. In our first result, we present the first sublinear, almost time-optimal, distributed algorithm for the single random walk problem in arbitrary networks that runs in time $\tilde{O}(\sqrt{\ell D})$ with high probability [1], where ℓ is the length of the walk. Our algorithm is randomized (Las Vegas type, i.e., it always outputs the correct result, but the running time claimed is with high probability). It is straightforward to generalize the results to a $\mathcal{CONGEST}(B)$ model, where $O(B)$ bits can be transmitted in a single time step across an edge.

The high-level idea behind our algorithm is to "prepare" a few short walks in the beginning and carefully stitch these walks together later as necessary. If there are not enough short walks, we construct more of them on the fly. We overcome a key technical problem by showing how one can perform many short walks in parallel without causing too much congestion.

Our algorithm exploits certain key properties of random walks. The key property is a bound on the number of times any node is visited in an ℓ-length walk, for any length $\ell = O(m^2)$. We prove that w.h.p. any node x is visited at most $\tilde{O}(d(x)\sqrt{\ell})$ times, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x). We then show that if only certain ℓ/λ special points of the walk (called as *connector points*) are observed, then any node is observed only $\tilde{O}(d(x)\sqrt{\ell}/\lambda)$ times. The algorithm starts with all nodes performing short walks (of length uniformly random in the range λ to 2λ for appropriately chosen λ) efficiently simultaneously; here the randomly chosen

[1] Throughout this thesis, "with high probability (whp)" means with probability at least $1 - 1/n^{\Omega(1)})$, where n is the number of nodes in the network.

lengths play a crucial role in arguing about a suitable spread of the connector points. Subsequently, the algorithm begins at the source and carefully stitches these walks together till ℓ steps are completed.

A Tight Lower Bound for Random Walk Computation. We also show a tight unconditional lower bound on the time complexity of distributed random walk computation. Specifically, we show that for any n, D, and $D \leq \ell \leq (n/(D^3 \log n))^{1/4}$, performing a random walk of length $\Theta(\ell)$ on an n-node network of diameter D requires $\Omega(\sqrt{\ell D} + D)$ time. This bound is *unconditional*, i.e., it holds for any (possibly randomized) algorithm. To the best of our knowledge, this is the first lower bound that the diameter plays a role of multiplicative factor. Our bound shows that our algorithm is time optimal.

Extensions. We also extend the result to give algorithms for computing k random walks (from any k sources —not necessarily distinct) in $\tilde{O}\left(\min(\sqrt{k\ell D} + k, k + \ell)\right)$ rounds. Computing k random walks is useful in many applications such as the one we present below on decentralized computation of mixing time and related parameters. While the main requirement of our algorithms is to just obtain the random walk samples (i.e. the end point of the ℓ step walk), our algorithms can regenerate the entire walks such that each node knows its position(s) among the ℓ steps. Our algorithm can be extended to do this in the same number of rounds.

We finally present extensions of our algorithm to perform random walk according to the Metropolis-Hastings [75, 113] algorithm, a more general type of random walk with numerous applications (e.g., [155]). The Metropolis-Hastings algorithm gives a way to define transition probabilities so that a random walk converges to any desired distribution. An important special case, is when the distribution is uniform.

Applications. Our faster distributed random walk algorithm can be used in speeding up distributed applications where random walks arise as a subroutine. Such applications include distributed construction of expander graphs, checking whether a graph is an expander, construction of random spanning trees, and random-walk based search (we refer to [44] for details). Here, we present two key applications:

(1) *A Fast Distributed Algorithm for Random Spanning Trees (RST):* We give a $\tilde{O}(\sqrt{m}D)$ time distributed algorithm (cf. Section 2.5.1) for uniformly sampling a random spanning tree in an arbitrary undirected (unweighted) graph (i.e., each spanning tree in the underlying network has the same probability of being selected). (m denotes the number of edges in the graph.) Spanning trees are fundamental network primitives and distributed algorithms for various types of spanning trees such as minimum spanning tree (MST), breadth-first spanning tree (BFS), shortest path tree, shallow-light trees etc., have been studied extensively in the literature [128]. However, not much is known about the distributed complexity of the random spanning tree problem. The centralized case has been studied for many decades, see e.g., the recent work of [83] and the references therein; also see the recent work of Goyal et al. [69] which gives nice applications of RST to fault-tolerant routing and constructing expanders. In the distributed context, the work of Bar-Ilan and Zernik [15] give a distributed RST algorithm for two special cases, namely that of a complete graph (running in constant time) and a synchronous ring (running in $O(n)$ time). The work of [12] give a self-stablizing distributed algorithm for constructing a RST in a wireless ad hoc network and mentions that RST is more resilient to transient failures that occur in mobile ad hoc networks.

Our algorithm works by giving an efficient distributed implementation of the well-known Aldous-Broder random walk algorithm [5, 22] for constructing a RST.

(2) *Decentralized Computation of Mixing Time.* We present a fast decentralized algorithm for estimating mixing time, conductance and spectral gap of the network

(cf. Section 2.5.2). In particular, we show that given a starting point x, the mixing time with respect to x, called τ^x_{mix}, can be estimated in $\tilde{O}(n^{1/2}+n^{1/4}\sqrt{D\tau^x_{mix}})$ rounds. This gives an alternative algorithm to the only previously known approach by Kempe and McSherry [84] that can be used to estimate τ^x_{mix} in $\tilde{O}(\tau^x_{mix})$ rounds.[2] To compare, we note that when $\tau^x_{mix} = \omega(n^{1/2})$ the present algorithm is faster (assuming D is not too large).

1.3.3 Related Work

The general high-level idea of using a few short walks in the beginning (executed in parallel) and then carefully stitch these walks together later as necessary was introduced in [40] to find random walks on data streams with the main motivation of finding PageRank. However, the two models have very different constraints and motivations and hence the subsequent techniques used here and in [40] are very different. Recently, Sami and Twigg [134] consider lower bounds on the communication complexity of computing stationary distribution of random walks in a network. Although, their problem is related to our problem, the lower bounds obtained do not imply anything in our setting. Other recent works involving multiple random walks in different settings include Alon et. al. [7], Elsässer et. al. [56], and Cooper et al. [36].

The work of [63] discusses spectral algorithms for enhancing the topology awareness, e.g., by identifying and assigning weights to critical links. However, the algorithms are centralized, and it is mentioned that obtaining efficient decentralized algorithms is a major open problem. Our algorithms are fully decentralized and based on performing random walks, and so more amenable to dynamic and self-organizing networks.

[2]Note that [84] in fact do more and give a decentralized algorithm for computing the top k eigenvectors of a weighted adjacency matrix that runs in $O(\tau_{mix}\log^2 n)$ rounds if two adjacent nodes are allowed to exchange $O(k^3)$ messages per round, where τ_{mix} is the mixing time and n is the size of the network.

1.4 How fast are approximation algorithms on distributed networks?

1.4.1 Problem

Distributed Algorithms, Approximation, and Hardness. Much of the initial research focus in the area of distributed computing was on designing algorithms for solving problems exactly, e.g., distributed algorithms for spanning tree (ST), minimum spanning tree (MST), and shortest paths are well-known [128, 110]. Over the last few years, there has been interest in designing distributed algorithms that provide approximate solutions to problems. This area is known as *distributed approximation*. One motivation for designing such algorithms is that they can run faster or have better communication complexity albeit at the cost of providing suboptimal solution. This can be especially appealing for resource-constrained and dynamic networks (such as sensor or peer-to-peer networks). For example, there is not much point in having an optimal algorithm in a dynamic network if it takes too much time, since the topology could have changed by that time. For this reason, in the distributed context, such algorithms are well-motivated even for network optimization problems that are not NP-hard, e.g., minimum spanning tree, shortest paths etc. There is a large body of work on distributed approximation algorithms for various classical graph optimization problems (e.g., see the surveys by Elkin [52] and Dubhashi et al. [51], and the work of [86] and the references therein).

While a lot of progress has been made in the design of distributed approximation algorithms, the same has not been the case with the theory of lower bounds on the approximability of distributed problems, i.e., *hardness of distributed approximation*. There are some inapproximability results that are based on lower bounds on the time complexity of the exact solution of certain problems and on integrality of the objective functions of these problems. For example, a fundamental result due to Linial [103] says that 3-coloring an n-vertex ring requires $\Omega(\log^* n)$ time. In particular, it

implies that any 3/2-approximation protocol for the vertex-coloring problem requires $\Omega(\log^* n)$ time. On the other hand, one can state inapproximability results assuming that vertices are computationally limited; under this assumption, any NP-hardness inapproximability result immediately implies an analogous result in the distributed model. However, the above results are not interesting in the distributed setting, as they provide no new insights on the roles of locality and communication [55].

There are but a few significant results currently known on the hardness of distributed approximation. Perhaps the first important result was presented for the MST problem by Elkin in [55]. Specifically, he showed strong *unconditional* lower bounds (i.e., ones that do not depend on complexity-theoretic assumptions) for distributed approximate MST (more on this result below). Later, Kuhn, Moscibroda, and Wattenhofer [93] showed lower bounds on time approximation trade-offs for several problems.

Distributed Verification. The above discussion summarized two major research aspects in distributed computing, namely studying distributed algorithms and lower bounds for (1) exact and (2) approximate solutions to various problems. The third aspect — that turns out to have remarkable applications to the first two — called *distributed verification*, is one of the main subjects of this thesis. In distributed verification, we want to efficiently check whether a given subgraph of a network has a specified property via a distributed algorithm[3]. Formally, given a graph $G = (V, E)$, a subgraph $H = (V, E')$ with $E' \subseteq E$, and a predicate Π, it is required to decide whether H satisfies Π (i.e., when the algorithm terminates, every node knows whether H satisfies Π). The predicate Π may specify statements such as "H is connected" or "H is a spanning tree" or "H contains a cycle". (Each vertex in G knows which of its incident edges (if any) belong to H.) The goal is to study bounds on the

[3]Such problems have been studied in the sequential setting, e.g., Tarjan [141] studied verification of MST.

time complexity of distributed verification. The time complexity of the verification algorithm is measured with respect to parameters of G (in particular, its size n and diameter[4] D), independently from H.

We note that verification is different from construction problems, which have been the traditional focus in distributed computing. Indeed, distributed algorithms for constructing spanning trees, shortest paths, and other problems have been well studied ([128, 110]). However, the corresponding verification problems have received much less attention. To the best of our knowledge, the only distributed verification problem that has received some attention is the MST (i.e., verifying if H is a MST); the recent work of Kor et al. [90] gives a $\Omega(\sqrt{n}/B + D)$ *deterministic* lower bound on distributed verification of MST, where D is the diameter of the network G. That paper also gives a matching upper bound (see also [91]). Note that distributed *construction* of MST has rather similar lower and upper bounds [129, 62]. Thus in the case of the MST problem, verification and construction have the same time complexity. We later show that the above result of Kor et al. is subsumed by the results of this thesis, as we show that verifying *any* spanning tree takes so much time, even when we allow one-sided error randomization.

Therefore, while computing a spanning tree can be done in $O(D)$ time, even with single-bit messages (as one can simply breadth-first-search the graph in time $O(D)$), *verifying* whether an existing spanning tree indeed is a correct spanning tree needs $\Omega(\sqrt{n} + D)$ time. (Thus, in contrast to traditional non-distributed complexity, verification is harder than computation in the distributed world!)

Our result is more general, as we show interesting lower and upper bounds (these are almost tight) for a whole selection of verification problems. Furthermore, we show

[4]The length of a path p in G is the number of edges it contains. The *distance* between two vertices is the length of the shortest path connecting them. The *diameter* D of G is the maximum distance between any two vertices of G.

a key application of studying such verification problems to proving strong unconditional time lower bounds on exact and approximate distributed computation for many classical problems.

Motivations. The study of distributed verification has two main motivations. The first is understanding the complexity of verification versus construction. This is obviously a central question in the traditional RAM model, but here we want to focus on the same question in the distributed model. Unlike in the centralized setting, it turns out that verification is *not* in general easier than construction in the distributed setting! In fact, as was indicated earlier, distributively verifying a spanning tree turns out to be harder than constructing it in the worst case. Thus understanding the complexity of verification in the distributed model is also important. Second, from an algorithmic point of view, for some problems, understanding the verification problem can help in solving the construction problem or showing the inherent limitations in obtaining an efficient algorithm. In addition to these, there is yet another motivation that emerges from this work: We show that distributed verification leads to showing *strong unconditional lower bounds on distributed computation (both exact and approximate)* for a variety of problems, many hitherto unknown. For example, we show that establishing a lower bound on the spanning connected subgraph verification problem leads to establishing lower bounds for the minimum spanning tree, shortest path tree, minimum cut etc. Hence, studying verification problems may lead to proving hardness of approximation as well as lower bounds for exact computation for new problems.

1.4.2 Results

In this thesis, our main contributions are two fold. First, we initiate a systematic study of *distributed verification*, and give almost tight uniform lower bounds on the running time of distributed verification algorithms for many fundamental problems.

Second, we make progress in establishing strong hardness results on the distributed approximation of many classical optimization problems. Our lower bounds also apply seamlessly to exact algorithms. We next state our main results (the precise theorem statements are in the respective sections as mentioned below).

1. Distributed Verification. We show a lower bound of $\Omega(\sqrt{n/(B\log n)}+D)$ for many verification problems in the B model, including *spanning connected subgraph*, *s-t connectivity*, *cycle-containment*, *bipartiteness*, *cut*, *least-element list*, and $s-t$ *cut*. These bounds apply to *randomized* algorithms as well, and clearly hold also for asynchronous networks. Moreover, it is important to note that our lower bounds apply even to graphs of small diameter ($D = O(\log n)$). (Indeed, the problems studied in this thesis are "global" problems, i.e., the network diameter of G imposes an inherent lower bound on the time complexity.)

Additionally, we show that another fundamental problem, namely, the spanning tree verification problem (i.e., verifying whether H is a spanning tree) has the same lower bound of $\Omega(\sqrt{n/(B\log n)}+D)$. However, this bound applies to only *one-sided error* randomized algorithms (see Chapter 5 for details). This result strengthens the deterministic lower bound result of MST verification by Kor et al. [90] in that it gives a lower bound even for verifying a spanning tree and holds for randomized algorithms. Moreover, we note the interesting fact that although finding a spanning tree (e.g., a breadth-first tree) can be done in $O(D)$ rounds [128], verifying if a given subgraph is a spanning tree requires $\tilde{\Omega}(\sqrt{n}+D)$ rounds! Thus the verification problem for spanning trees is harder than its construction in the distributed setting. This is in contrast to this well-studied problem in the centralized setting.

Our lower bounds are almost tight as we show that there exist algorithms that run in $O(\sqrt{n}\log^* n+D)$ rounds (assuming $B = O(\log n)$) for all the verification problems addressed here.

Table 1: Lower bounds of randomized α-approximation algorithms on graphs of various diameters. Bounds in the first column are for the MST and shortest path tree problems [55] while those in the second column are for these problems and many problems listed in Fig. 11. We note that these bounds almost match the $O(\sqrt{n}\log^* n + D)$ upper bound for the MST problem [62, 96] and are independent of the approximation factor α.

Diameter D	Previous lower bound for MST and shortest-path tree [55] (for exact algorithms, use $\alpha = 1$)	New lower bound for MST, shortest-path tree and all problems in Fig. 11.
n^δ, $0 < \delta < 1/2$	$\Omega(\sqrt{\frac{n}{\alpha B}})$	$\Omega(\sqrt{\frac{n}{B}})$
$\Theta(\log n)$	$\Omega(\sqrt{\frac{n}{\alpha B \log n}})$	$\Omega(\sqrt{\frac{n}{B \log n}})$
Constant ≥ 3	$\Omega((\frac{n}{\alpha B})^{\frac{1}{2}-\frac{1}{2D-2}})$	$\Omega((\frac{n}{B})^{\frac{1}{2}-\frac{1}{2D-2}})$
4	$\Omega((\frac{n}{\alpha B})^{1/3})$	$\Omega((\frac{n}{B})^{1/3})$
3	$\Omega((\frac{n}{\alpha B})^{1/4})$	$\Omega((\frac{n}{B})^{1/4})$

2. Bounds on Hardness of Distributed Approximation. An important consequence of our verification lower bound is that it leads to lower bounds for exact and approximate distributed computation. We show the unconditional time lower bound of $\Omega(\sqrt{n/(B\log n)} + D)$ for approximating many optimization problems, including MST, shortest $s-t$ path, shortest path tree, and minimum cut (Section 5.4). The important point to note is that the above lower bound applies for *any* approximation ratio $\alpha \geq 1$. Thus the same bound holds for exact algorithms also ($\alpha = 1$). All these hardness bounds hold for randomized algorithms. As in our verification lower bounds, these bounds apply even to graphs of small ($O(\log n)$) diameter. Figure 1 summarizes our lower bounds for various diameters.

Our results improve over previous ones (e.g., Elkin's lower bound for approximate MST and shortest path tree [55]) and subsumes some well-established exact bounds (e.g., Peleg and Rubinovich lower bound for MST [129]) as well as shows new strong bounds (both for exact and approximate computation) for many other problems (e.g., minimum cut), thus answering some questions that were open earlier (see the survey by Elkin [52]).

The new lower bound for approximating MST simplifies and improves upon the

previous $\Omega(\sqrt{n/(\alpha B \log n)}+D)$ lower bound by Elkin [55], where α is the approximation factor. [55] showed a *tradeoff* between the running time and the approximation ratio of MST. Our result shows that approximating MST requires $\Omega(\sqrt{n/(B \log n)}+D)$ rounds, regardless of α. Thus our result shows that there is actually no trade-off, since there can be no distributed approximation algorithm for MST that is significantly faster than the current exact algorithm [96, 54], for any approximation factor $\alpha > 1$.

1.5 How fast can we compute skylines on data streams and distributed networks?

1.5.1 Problem

The skyline of a d-dimensional dataset is the set of points (tuples) that are not *dominated* by any other point, where we say that a point p dominates another point p' if the coordinate of p on each dimension is not smaller than that of p', and strictly larger on at least one dimension. A popular example is a hotel reservation system. Consider a hypothetical scenario in which a tourist is searching for a hotel that both is cheap and has high quality. Although most hotels that have higher quality tend to be more expensive, there could be real-life instances in which a hotel A is more expensive than a hotel B but B has better quality than A. Clearly, in this scenario, inferior hotels such as A should not be shown to this tourist. For example, Figure 5 shows the first 5 query results of hotels in Lyon from August 24 to August 28, 2009 (during the time of the VLDB'09 conference), when we search `hotels.com` in March 2009. Should `hotels.com` support skyline query and someone execute the query `[Select *, From Hotels, Skyline of Price min, Guest_Rating max]`, then hotels 2, 4, 5 should not be shown as they are dominated by hotel 1 (the latter has better guest rating and lower price); Only 1 and 3 should be on the skyline, as one has a better guest rating and the other has a lower price.

Since a database is usually too large to be stored in the main memory, skyline algorithms are external [21] in the sense that data resides on the external memory

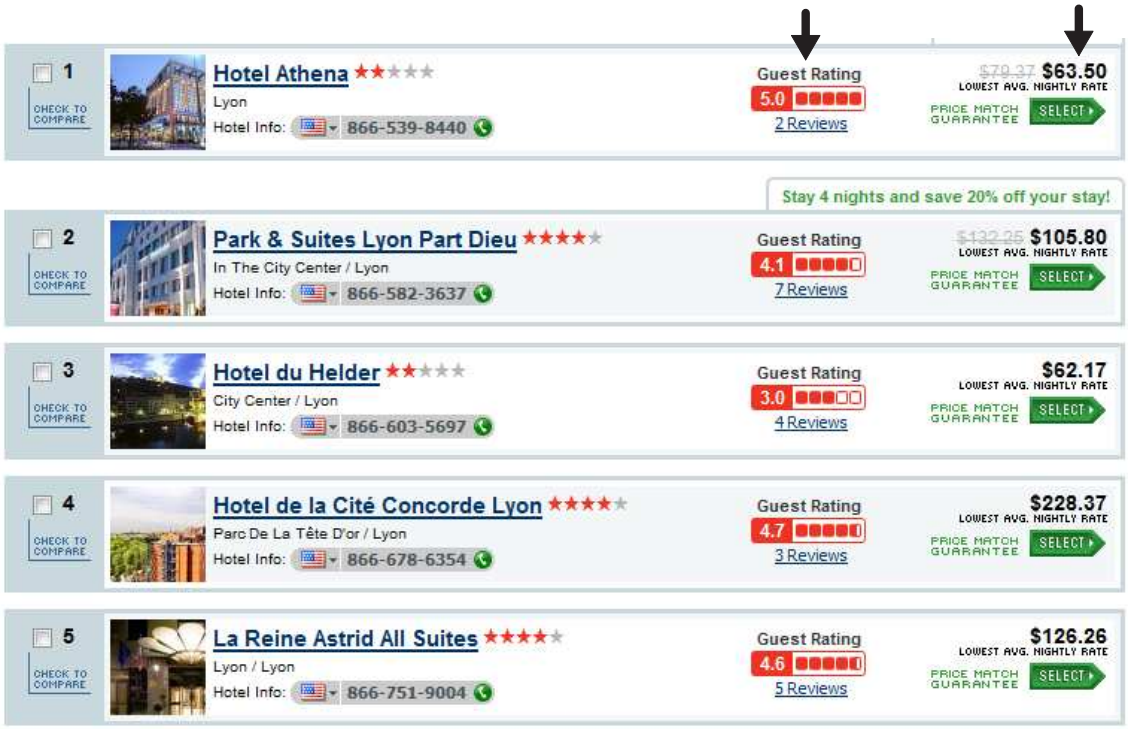


Figure 5: Search result from `hotels.com`.

(i.e., disk) while its processing happens in the main memory. Skyline algorithms can be classified into two categories: with and without pre-processing. Pre-processing such as indexing [123] and sorting [32] helps speed up the skyline computation, but maintaining indices over a large number of dimensions could be computationally intensive. Without pre-processing, skyline algorithms have to take at least one pass over the database for computing the skyline and therefore are often slower than those with pre-processing. However, their flexibility and wider applicability makes them attractive alternatives in many application scenarios [65]. In this thesis, we focus on the skyline algorithms without pre-processing.

In this thesis, we appeal to the concept of optimizing the worst case behavior of skyline algorithms. There are many real-time applications in which one would be interested in upper-bounding the times taken in skyline computation and minimizing its variance. Consider a flight booking site where the site searches for results, when

presented with a query, and displays a skyline set of options. If the time taken for this page to load is longer than usual, the user is likely to abandon the search and try one of many other such sites. Another example is skyline queries on stock markets, where prices change very rapidly. If a user (or automatic trading software on behalf of the user) wants to perform a transaction based on the results from the skyline query, presenting the results reliably fast is essential.

In this thesis, we propose a set of multi-pass data streaming algorithms that can compute the skyline of a massive database with strong worst-case performance guarantees (motivated above). The data stream in this context refers to the data items in the massive database (residing on disks) that are read into and processed through the main memory in a stream fashion. Our algorithms have a salient feature: They are not sensitive (performance wise) to the order in which the data items show up in the stream. Therefore, we can simply read the data items out from the database in the "physically sequential order" in the sense that disk heads only need to move in one direction during each stream pass. Since the seek and rotational delays of a disk are orders of magnitude larger than the transmission delay, this feature may translate into significant performance improvements in real-world systems.

Even with physically sequential accesses of disks, however, each pass over the database is still quite time-consuming. Therefore, our skyline algorithms need to minimize the number of such passes in order to have good performance. This objective is achieved using randomization[5], which turns out to be a powerful tool in quickly eliminating a large number of non-skyline points from consideration after each pass.

1.5.2 Results

The contributions of this thesis can be summarized as follows.

[5]Our technique uses randomness in the algorithm but does not assume any distribution on the input.

- We formalize the multi-pass streaming model (implicitly with physically sequential accesses of disks) for the skyline problem. We are interested in worst case analysis, in terms of random and sequential I/O's and comparison operations. We prove a simple yet instructive performance lower bound under this model.

- Our key contribution is a randomized multi-pass streaming algorithm, RAND. We present two versions, one with and one without fixed window, and prove their theoretical worst-case performance guarantees. These performance guarantees, combined with the aforementioned lower bound, shows that RAND algorithms are near-optimal.

- We extend RAND to the distributed networks model. We show an algorithm that runs in $\tilde{O}(m)$ time which is almost optimal. RAND can also be extended to other settings: (1) It extends to a deterministic variant that works for the two-dimensional case, and (2) works even for partially-ordered domains.

- We perform extensive experiments on real and synthetic data, which show that RAND is comparable to the state-of-the-art skyline algorithms in various performance metrics. We also show that, with certain perturbations of the data orders, the performance of the other algorithms degrade considerably while RAND is robust to such changes.

1.5.3 Related Work

Skyline computation, previously known as Pareto sets, admissible points, and maximal vectors, has been extensively studied in the theory and mathematics community since 1960s (see, e.g., [17, 94, 112]). However, they assume that the whole input data can be fit in the internal memory. Therefore, these algorithms do not scale well to large databases. In particular, any scalable algorithm should be *external*. The problem in this setting, and the name *skyline*, were introduced to the database community

by Börzsönyi et al. [21] and has been studied extensively since then.

As mentioned in the introduction, external skyline algorithms can be classified into two categories: with and without pre-processing. Each category has its own advantages and disadvantages. Algorithms presented in this thesis are in the latter category. We review some state-of-the-art algorithms in this category that we compare against: BNL and LESS. Both algorithms serve as good benchmarks for comparison as they perform well with respect to different parameters.

BNL [21] operates with a memory window of size w. It makes several passes over the data, each time storing the first w points that are undominated. While reading the disk, any point that is dominated is eliminated so that it is never read in the future. BNL has a timestamping mechanism that is used to determine when a point is in the skyline and when all points it dominates have been eliminated. It continues to make passes over the data until all skyline points have been obtained. BNL remains to be a classic algorithm in that several other algorithms use BNL or a modification of it as a subroutine.

LESS [65] is an extension of the SFS algorithm [32]. SFS assumes that the data is pre-processed by sorted according to a scoring function. Once sorted, a BNL-like filtering procedure can then be performed on the data to get the skyline points. Sorting the data gives the desirable property that, as soon as a point gets added to the buffer, it can be output as being in the skyline and does not need any timestamping overhead. The authors of [32] suggest the use of the entropy function to efficiently eliminate many tuples. LESS eliminate some more points while sorting and integrates the final pass of the external sorting phase with the first filter pass.

Another category of skyline algorithms that we do not consider here are those with pre-processing. The main feature of these algorithms is that they can compute skyline without going through the whole input data; thus, they are *progressive*. Most of the algorithms in this category are index-based and exploit R-tree and its variations to

obtain good performances. The first algorithm in this category is the nearest neighbor (NN) algorithm in [92] and the state-of-the-art algorithm in this category is BBS [123] which is I/O-optimal. As mentioned in the introduction, we do not consider this category in this thesis.

Several other variants have been considered. Since it is not possible to list a complete survey of all papers, we mention a few here. Algorithms using the bitmap method [138] and for partially-ordered attributes to capture dynamic user preferences [28, 27, 148, 133] , computing cardinality or exploiting low cardinality domains [30, 116], sliding window or time-series skyline queries [102, 140, 80, 154], distributed and super-peer architectures [14, 147, 157, 126], representative skylines [139], probabilistic skylines on uncertain data [127] have been studied.

There has been work on computing skylines in a streaming setting [102, 140, 80, 154]. However, these works look at single-pass streams under the sliding window model, whereas we are interested in multi-pass algorithms.

For works related to skyline computation on distributed networks, see Zhu et al. [157] for an excellent survey and description of many distributed models. Also see [39, 31, 47] and references therein.

1.6 How fast can we solve graph problems on data streams?

1.6.1 Problem

We are motivated by three fundamental questions that arise in three widely studied areas in theoretical computer science - streaming algorithms, communication complexity, and proof checking. The first question, which is of our main interest, is how efficient can space restricted streaming algorithms be. The second question, is whether the lower bound of a communication problem holds for every partition of the input. Finally, in proof checking, the question is how many (extra) bits are needed for the verifier to establish a proof in a restricted manner. Before elaborating on these

questions, we first describe an application that motivates our model.

Many big companies such as Amazon [1] and `salesforce.com` are currently offering *cloud computing* services. These services allow their users to use the companies' powerful resources for a short period of time, over the Internet. They also provide some softwares that help the users who may not have knowledge of, expertise in, or control over the technology infrastructure ("in the cloud") that supports them.[6] These services are very helpful, for example, when a user wants a massive computation over a short period of time.

Now, let us say that we want the cloud computer to do a simple task such as checking if a massive graph is strongly connected. Suppose that the cloud computer gets back to us with an answer "Yes" suggesting that the graph is strongly connected. What do we make of this? What if there is a bug in the code, or what if there was some communication error? Ideally one would like a way for the cloud to *prove* to us that the answer is correct. This proof might be long due to the massive input data; hence, it is impossible to keep everything in our laptop's main memory. Therefore, it is more practical to read the proof as a *stream* with a small working memory. Moreover, the proof should not be too long – one ideal case is when the proof is the input itself (in a specific order). This is the model considered in this thesis. Related models motivated by similar applications have also been studied by Li et al. [101, 153], Papadopoulos et al. [125], Goldwasser et al. [67], Chakrabarti et al. [25], and Cormode et al. [38].

We describe previous models studied specifically in the stream, computational complexity, and proof checking domains and contrast them with our model.

Data Streams: Recall the streaming model described in Section 1.1. The basic premise of streaming algorithms is that one is dealing with a humongous data set, too large to process in main memory. The algorithm has only sequential access to

[6] `http://www.ebizq.net/blogs/saasweek/2008/03/distinguishing_cloud_computing/`.

the input data; this is called a *stream*. In certain settings, it is acceptable to allow the algorithm to perform multiple passes over the stream. The general streaming algorithms framework has been studied extensively since the seminal work of Alon, Matias, Szegedy [8].

Models diverge in the assumptions made about what order the algorithm can access the input elements in. In the classic Finite State Automata model [137], the order of the data is set by the definition of the problem. Streaming models allow a richer class of order types. The most stringent restriction on the algorithm is to assume that the input sequence is presented to the algorithm in an adversarial order. A slightly more relaxed setting, that has also been widely studied is where the input is assumed to be presented in randomized order [26, 71, 72]. However, even a simple problem like finding median (which was considered in the earliest paper in the area by Munro and Patterson [117]) was shown recently [26] to require $\Omega(\log \log n)$ passes in both input orders if the space is bounded by $O(\text{polylog}\, n)$. In [76], one of the earliest paper in this area, it was shown that many graph problems require prohibitively large amount of space to solve. It is confirmed by the more recent result [58] that a huge class of graph problems cannot be solved efficiently in a few passes. Since then, new models have been proposed to overcome this obstruction. Feigenbaum et. al. [59] proposed a relaxation of the memory restriction in what is called the semi-stream model. Another input order suggested by Aggarwal et. al. [3] is that of receiving the input in some sorted order. In the classic Binary Decision Diagram [89] the order used is of *best oblivious*; i.e., the input is presented in the best manner for the problem but not necessarily for the problem instance.

Another model that has been considered is the W-Stream (write-stream) model [132, 46]. While the algorithm processes the input, it may also *write* a new stream to be read in the next pass.

We ask the following fundamental question:

If the input is presented in the best order possible, can we solve problems efficiently?

A precise explanation is reserved for the models in Chapter 7; however, intuitively, this means that the algorithm processing the stream can decide on a *rule* on the order in which the stream is presented. We call this the best-order stream model. For an example, if the rule adopted by the algorithm is to read the input in sorted order, then this is equivalent to the single pass sort stream model. Another example of a rule, for graphs presented as edge streams could be that the algorithm requires all edges incident on a vertex to be presented together. This is again equivalent to a graph stream model studied earlier called the incidence model (and corresponds to reading the rows of the adjacency matrix one after the other). A stronger rule could be that the algorithm asks for edges in some perfect matching followed by other edges. As we show in this thesis, this rule leads to checking if the graph has a perfect matching and as a consequence shows the difference between our model and the sort-stream model.

Communication Complexity: Another closely related model is the communication complexity model [152, 95]. Recall from Section 1.1 that, in the basic form of this model, two players, Alice and Bob, receive some input data and they want to compute some function together. The question is how much communication they have to make to accomplish the task. There are many variations of how the input is partitioned. The worst-case [97] and the best-case [124] partition models are two extreme cases that are widely studied over decades. The worst case asks for the partition that makes Alice and Bob communicate the most while the best case asks for the partition that makes the communication smallest. Moreover, even very recently, another variation where the input is partitioned according to some known distribution (see, e.g., [24]) was proposed. The main question is whether the lower bound of

a communication problem holds for almost every partition of the input, as opposed to holding for perhaps just a few atypical partitions.

The communication complexity version of our model (described in Chapter 7) asks the following similar question: Does the lower bound of a communication problem hold for *every* partition of the input? Moreover, our model can be thought of as a more extreme version of the best-case partition communication complexity. We explain this in more details in Chapter 7.

Proof Checking: From a complexity theoretic standpoint, our model can be thought of as the case of proof checking where a polylog-space verifier is allowed to read the proof as a stream; additionally, the proof must be the input itself in a different order.

The field of probabilistically checkable proofs (PCPs) [10, 11, 48] deals with a verifier querying the proof at very few points (even if the data set is large and thus the proof) and using this to guarantee the proof with high probability. While several variants of proof checking have been considered, we only state the most relevant ones. A result most related to our setting is by Lipton [105] where it was shown that membership proofs for NP can be checked by probabilistic logspace verifiers that have one-way access to the proof and use $O(\log n)$ random bits. This result almost answers our question except that the proof is not the reordered input and, more importantly, its size is not linear (but polynomial) in the size of the input which might be too large for many applications.

Another related result that compares streaming model with other models is by Feigenbaum et. al. [57] where the problem of testing and spot-checking on data streams is considered. They define sampling-tester and streaming-tester. A sampling-tester is allowed to sample some (but not all) of the input points, looking at them in any order. A streaming-tester, on the other hand is allowed to look at the entire input but only in a specific order. They show that some problems can be solved in

a streaming-tester but not by a sampling-tester, while the reverse holds for other problems. Finally, we note that our model (when we focus on massive graphs) might remind some readers of the problem of property testing in massive graphs [66]. Chakrabarti et al. [25] consider an annotation model for streaming proofs, again motivated by cloud computing services. Their model allows a helper to add additional bits to the stream to generate a proof to be presented to the verifier. In this model, the helper observes the stream concurrently with the algorithm. In follow up work to Chakrabarti et al. [25] and this thesis, Cormode et al. [38] consider a similar annotation model where the cloud and the verifier look at the stream input. Subsequently, the cloud service needs to provide a proof to the verifier about the specific problem, which may include the reordered stream and may include additional helper bits as well. The verifier still needs to work with small space though since the proof itself may be long.

Notice that in all of the work above, there are two common themes. The first is verification using *small space*. The second is some form of *limited access* to the input. The limited access is either in the form of sampling from the input, limited communication, or some restricted streaming approach. Our model captures both these aspects.

1.6.2 Results

In this thesis, we partially answer whether there are efficient streaming algorithms when the input is in the best order possible. We give a negative answer to this question for the deterministic case and show evidence of a positive answer for the randomized case. Our positive results are similar in spirit to those for the W-stream and Sort-stream models [3, 46, 132].

For the negative answer, we show that the space requirement is too large even

for the simple problem of checking if a given graph has a perfect matching deterministically. In contrast, this problem, as well as the connectivity problem, can be solved efficiently by randomized algorithms. We show similar results for other graph properties.

1.7 Organization of this thesis

In the first chapter, Chapter 2, we develop random walk algorithms and show their applications. In the next two chapters, Chapter 3 and 4 we develop ideas towards the proof that our algorithm for performing a random walk is optimal. In particular, in Chapter 3, we establish a connection between variations of communication complexity and distributed algorithm lower bounds. Then, in Section 4, we use this connection to prove the random walk lower bound. This concludes the results on distributed random walk computation. In Chapter 5, we show that our technique for proving the random walk lower bound can be applied to many other graph problems, in particular, approximating the MST and distance. In Chapter 6, we turn to the geometric algorithms for computing skylines where we show almost optimal algorithms for computing skyline on data streams and distributed networks. Finally, in Chapter 7, we explore the power of streaming graph algorithms by considering the best-order stream.

CHAPTER II

RANDOM WALK ALGORITHMS AND APPLICATIONS

In this chapter, we present the first sublinear, almost time-optimal, distributed algorithm for the single random walk problem (1-RW-DoS) in arbitrary networks that runs in time $\tilde{O}(\sqrt{\ell D})$ with high probability, where ℓ is the length of the walk (the precise theorem is stated in Section 2.1). The algorithm and analysis can be found in Section 2.1, 2.2, and 2.3. We will show that this algorithm is optimal in Chapter 4.

Recall that in this problem, we are given an arbitrary undirected, unweighted, and connected n–node network $G = (V, E)$ in the $\mathcal{CONGEST}$ model (defined in Section 1.1.1) and a source node $s \in V$. The goal is to devise a distributed algorithm such that, in the end, some node v outputs the ID of s, where v is a destination node picked according to the probability that it is the destination of a random walk of length ℓ starting at s.

In Section 2.4, we show extensions and variations of the above result. We give algorithms for computing k random walks (k-RW-DoS), generating the entire random walk so that every node knows their positions (k-RW-PoS), and performing random walk according to the Metropolis-Hastings [75, 113] algorithm, a more general type of random walk with numerous applications (e.g., [155]). Finally, in Section 2.5, we present two key applications, generating random spanning trees (RST) and computation of the mixing time.

Throughout this chapter, "with high probability (whp)" means with probability at least $1 - 1/n^{\Omega(1)}$, where n is the number of nodes in the network. We assume the standard (simple) random walk: in each step, an edge is taken from the current node v with probability proportional to $1/d(v)$ where $d(v)$ is the degree of v. Our goal is

to output a true random sample from the ℓ-walk distribution starting from s.

2.1 Algorithm for 1-RW-DoS

In this section we describe the algorithm to sample one random walk destination. We show that this algorithm takes $\tilde{O}(\sqrt{\ell D})$ rounds with high probability and extend it to other cases in the next sections. First, we make the following simple observation, which will be assumed throughout.

Observation 2.1. We may assume that ℓ is $O(m^2)$, where m is the number of edges in the network.

The reason is that if ℓ is $\Omega(m^2)$, the required bound of $\tilde{O}(\sqrt{\ell D})$ rounds is easily achieved by aggregating the graph topology (via upcast) onto one node in $O(m + D)$ rounds (e.g., see [128]). The difficulty lies in proving for $\ell = O(m^2)$.

A Slower algorithm: Let us first consider the slow version of the algorithm to highlight the fundamental idea used to achieve sub-linear time bound. The high-level idea is to perform "many" short random walks in parallel and later stitch them together as needed. In particular, we perform the algorithm in two phases, as follows.

In Phase 1, we perform η "short" random walks of length λ from each node v, where η and λ are some parameters whose values will be fixed in the analysis. This is done naively by forwarding η "coupons" having the ID of v, from v to random destinations[1], as follows.

1: Initially, each node v crate η messages (called coupons) $C_1, C_2, ..., C_\eta$ and write its ID on them.

2: **for** $i = 1$ to λ **do**

[1]The term "coupon" refers to the same meaning as the more commonly used term of "token" but we use the term coupon here and reserve the term token for the second phase.

3: This is the i-th iteration. Each node v does the following: Consider each coupon C held by v which is received in the $(i-1)$-th iteration. Note v picks a neighbor u uniformly at random and forward C to u after incrementing the counter on the coupon to i.

4: **end for**

At the end of the process, for each node v, there will be η coupons containing v's ID distributed to some nodes in the network. These nodes are the destinations of short walks of length λ starting at v.

For Phase 2, let us first consider an algorithm that is slightly incomplete to avoid some unnecessary details. Starting at source s, we "stich" some of λ-length walks prepared in Phase 1 together to form a longer walk. The algorithm starts from s and randomly picks one coupon distributed from s in Phase 1. This is equivalent to having every node holding coupons of s writing their IDs on the coupon and send the coupons back to s. Then s picks one of these coupons randomly and return the rest to the owners. (However, aggregating all coupons at s is inefficient. The better way to do this is to use the idea of *reservoir sampling*. We will develop an algorithm called Sample-Coupon to do this job efficiently later on.)

Let C be the sampled coupon and v be the destination node of C. The source s then sends a "token" to v and delete coupon C (so that C will not be sampled again next time). The process then repeats. That is, the node v currently holding the token samples one of the coupons it distributed in the previous phase and forwards the token to the destination of the sampled coupon, say v'. A crucial observation is that the walk of length λ used to distribute the corresponding coupons from s to v and from v to v' are independent random walks. Therefore, we can stich them to get a random walk of length 2λ. (This fact will be formally proved in the next section.) We therefore can generate a random walk of length $3\lambda, 4\lambda, ...$ by repeating this process. We do this until we have completed more than $\ell-\lambda$ steps. Then, we complete the rest

of the walk by doing the naive random walk algorithm. The algorithm for Phase 2 is thus the following.

1: The source node s creates a message called "token" which contains the ID of s
2: **while** Length of walk completed is at most $\ell - \lambda$ **do**
3: Let v be the node that is currently holding the token.
4: v calls SAMPLE-COUPON(v) to sample one of the coupons distributed by v uniformly at random. Let C be the sampled coupon.
5: Let v' be the node holding coupon C. (ID of v' is written on C.)
6: v sends the token to v' and v' deletes C so that C will not be sampled again.
7: The length of walk completed has now increased by λ.
8: **end while**
9: Walk naively (i.e., forward the token to a random neighbor) until ℓ steps are completed.
10: A node holding the token outputs the ID of s

Figure 6 illustrates the idea of this algorithm. To understand the intuition behind this (incomplete) algorithm, let us analyze its running time. First, we claim that Phase 1 needs $\tilde{O}(\eta\lambda)$ rounds with high probability. This is because if we send out one coupon from each node at the same time, each node should receive one coupon in the average case. In other words, there is no congestion (i.e., no two coupons are sent through the same edge). Therefore sending out one coupon from each node for λ steps will take $O(\lambda)$ rounds in expectation and the time becomes $O(\eta\lambda)$ for η coupons. This argument can be modified to show that we need $\tilde{O}(\eta\lambda)$ rounds with high probability. (The full proof will be provided in Lemma 2.3 in the next section.) We will also show that SAMPLE-COUPON can be done in $O(D)$ rounds and it follows that Phase 2 needs $O(D \cdot \ell/\lambda)$ rounds. Therefore, the algorithm needs $\tilde{O}(\eta\lambda + D \cdot \ell/\lambda)$ which is $\tilde{O}(\sqrt{\ell D})$ when we set $\eta = 1$ and $\lambda = \sqrt{\ell D}$.

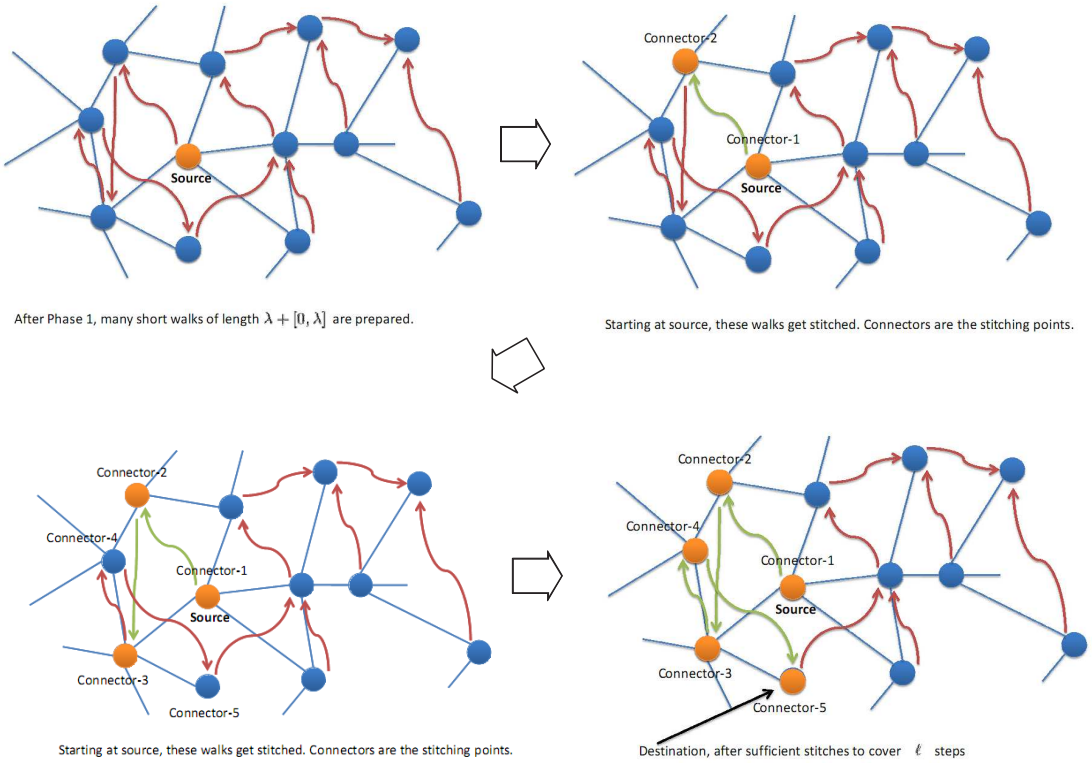


Figure 6: Figure illustrating the Algorithm of stitching short walks together.

The reason the above algorithm for Phase 2 is slightly incomplete is that it is possible that η coupons are not enough: We might forward the token to some node v many times in Phase 2 and all coupons distributed by v in the first phase are deleted. To cope with this problem, we develop an algorithm called Send-More-Coupons to distribute more coupons. In particular, when there is no coupon of v left in the network and v wants to sample a coupon, it calls Send-More-Coupons to send out η new coupons to random nodes. (Send-More-Coupons gives the same result as Phase 1 but the algorithm will be different in order to get a good running time.) In particular, we insert the following lines between Line 4 and 5 of the previous algorithm.

1: **if** $C =$ null (all coupons from v have already been deleted) **then**
2: v calls Send-More-Coupons(v, η, λ) (Distribute η new coupons. These coupons are forward for λ rounds.)
3: v calls Sample-Coupon(v) and let C be the returned coupon
4: **end if**

To complete this algorithm we now describe Sample-Coupon and Send-More-Coupons. The main idea of algorithm Sample-Coupon is to sample the coupons through a BFS tree from the leaves upward to the root. We allow each node to send only one coupon to its parent to avoid congestion. That is, in each round some node u will receive some coupons from its children (at most one from each child). Let these children be $u_1, u_2, ..., u_q$. Then, u picks one of these coupons and send to its parent. To ensure that u picks a coupon from uniform distribution, it picks the coupon received from u_i with probability proportional to the number of coupons in the subtree rooted at u_i. The precise statement of this algorithm can be found in Algorithm 2.1. The correctness of this algorithm (i.e., it outputs a coupon from uniform probability) will be proved in the next section (cf. Claim 2.8).

Algorithm 2.1 SAMPLE-COUPON(v)

Input: Starting node v.

Output: A node sampled from among the nodes holding the coupon of v

1: Construct a Breadth-First-Search (BFS) tree rooted at v. While constructing, every node stores its parent's ID. Denote such tree by T.
2: We divide T naturally into levels 0 through D (where nodes in level D are leaf nodes and the root node v is in level 0).
3: Every node u that holds some coupons of v picks one coupon uniformly at random. Let C_0 denote such coupon and let x_0 denote the number of coupons u has. Node u writes its ID on coupon C_0.
4: **for** $i = D$ down to 0 **do**
5: Every node u in level i that either receives coupon(s) from children or possesses coupon(s) itself do the following.
6: Let u have q coupons (including its own coupons). Denote these coupons by $C_0, C_1, C_2, \ldots, C_q$ and let their counts be $x_0, x_1, x_2, \ldots, x_q$. Node u samples one of C_0 through C_q, with probabilities proportional to the respective counts. That is, for any $1 \leq j \leq q$, C_j is sampled with probability $\frac{x_j}{x_0+x_1+\ldots+x_q}$.
7: The sampled coupon is sent to the parent node (unless already at root) along with a count of $x_0 + x_1 + \ldots + x_q$ (the count represents the number of coupons from which this coupon has been sampled).
8: **end for**
9: The root outputs the ID of the owner of the final sampled coupon (written on such coupon).

The SEND-MORE-COUPONS algorithm does essentially the same as what we did in Phase 1 with only one exception: Since this time we send out coupons from only one node, we can avoid congestions by combining coupons delivered on the same edge in each round. This algorithm is described in Algorithm 2.2, Part 1. (We will describe Part 2 later after we explain how to speed up the algorithm).

The analysis in the next section shows that SEND-MORE-COUPONS are called at most $\ell/(\eta\lambda)$ times in the worst case and it follows that the algorithm above takes time $\tilde{O}(\ell^{2/3}D^{1/3})$.

Faster algorithm: We are now ready to introduce the second idea which will complete the algorithm. To speed up the above slower algorithm, we randomly pick the length of each short walk uniformly at random in range $[\lambda, 2\lambda - 1]$, instead of fixing

Algorithm 2.2 SEND-MORE-COUPONS(v, η, λ)

Part 1: Distribute η coupons for λ steps.

1: The node v constructs η (identical) messages containing its ID.
2: **for** $i = 1$ to λ **do**
3: Each node u does the following:
4: - For each coupon C held by u, pick a neighbor z uniformly at random as a receiver of C.
5: - For each neighbor z of u, send ID of v and the number of coupons that z is picked as a receiver, denoted by $c(u, v)$.
6: - For each neighbor z of u, upon receiving ID of v and $c(u, v)$, constructs $c(u, v)$ coupons, each contains the ID of v.
7: **end for**

Part 2: Each coupon is now forwarded for λ steps. These coupons are now extended probabilistically further by r steps where each r is independent and uniform in the range $[0, \lambda - 1]$.

1: **for** $i = 0$ to $\lambda - 1$ **do**
2: For each coupon, independently with probability $\frac{1}{\lambda - i}$, stop sending the message further and save the ID of the source node (in this event, the node with the message is the destination). For each coupon that is not stopped, each node picks a neighbor correspondingly and sends the coupon forward as before.
3: **end for**
4: At the end, each destination node knows the source ID as well as the number of time the corresponding coupon is forwarded.

it to λ. The reason behind this is that we want every node in the walk to have some probability to take part in token forwarding in Phase 2. We called these nodes (that forward the token at some point in Phase 2) the *connectors*.

For example, consider running our random walk algorithm on a star network starting at the center and let $\lambda = 2$. If all short walks have length two then the center will always forward the token to itself in Phase 2. In other words, the center is the only connector and thus will appear as a connector $\ell/2$ times. This is undesirable since we have to prepare many walks from the center. In contrast, if we random the length of each short walk between two and three then the number of times that the center is a connector is $\ell/4$ in expectation. (To see this, observe that, regardless of where the token currently is, the the token will be forwarded to the center with

probability 1/2.)

In the next section, we will show an important property of the random walk which says that a random walk of length ℓ will visit each node v a few number of times compared to its degree, namely $\tilde{O}(\sqrt{\ell}\deg(v))$ times. We then use the above modification to claim that each node will be visited as a connector a few number of times too, namely $\tilde{O}(\sqrt{\ell}\deg(v)/\lambda)$ times. This implies that each node does not have to prepare many short walks which leads to the improved running time.

To do this modification, we need to modify Phase 1 and Send-More-Coupons. For Phase 1, we simply change the length of each short walk to $\lambda + r$ where r is a random integer in $[0, \lambda - 1]$. This modification is shown in Algorithm 2.3. A very slight change is also made on Phase 2. For a technical reason, we also prepare $\eta \deg(v)$ coupons from each node in Phase 1, instead of previously η coupons. Our analysis in the next section shows that this modification still needs $\tilde{O}(\eta\lambda)$ rounds as before.

To modify Send-More-Coupons, we add Part 2 to the algorithm (as in Algorithm 2.2) where we keep forwarding each coupon with some probability. It can be shown by a simple calculation that the number of steps each coupon is forwarded is uniformly between λ and $2\lambda - 1$.

We now have the complete description of the algorithm and are ready to show the analysis.

2.2 Analysis of Single-Random-Walk *algorithm*

We divide the analysis into four parts. First, we show the correctness of Algorithm Single-Random-Walk. (The proofs will be shown in subsequent sections.)

Lemma 2.2. *Algorithm* Single-Random-Walk *solves* 1*-RW-DoS. That is, for any node v, after algorithm* Single-Random-Walk *finishes, the probability that v outputs the ID of s is equal to the probability that it is the destination of a random walk of length ℓ starting at s.*

Algorithm 2.3 SINGLE-RANDOM-WALK(s, ℓ)

Input: Starting node s, and desired walk length ℓ and parameters λ and η.
Output: A destination node of the random of length ℓ output the ID of s.
Phase 1: Generate short walks by coupons distribution. Each node v performs $\eta \deg(v)$ random walks of length $\lambda + r_i$ where r_i (for each $1 \leq i \leq \eta \deg(v)$) is chosen independently and uniformly at random in the range $[0, \lambda - 1]$. At the end of the process, there are $\eta \deg(v)$ (not necessarily distinct) nodes holding a "coupon" containing the ID of v.

1: **for** each node v **do**
2: Generate $\eta \deg(v)$ random integers in the range $[0, \lambda - 1]$, denoted by $r_1, r_2, ..., r_{\eta \deg(v)}$.
3: Construct $\eta \deg(v)$ messages containing its ID and in addition, the i-th message contains the desired walk length of $\lambda + r_i$. We will refer to these messages created by node v as "coupons created by v".
4: **end for**
5: **for** $i = 1$ to 2λ **do**
6: This is the i-th iteration. Each node v does the following: Consider each coupon C held by v which is received in the $(i-1)$-th iteration. If the coupon C's desired walk length is at most i, then v keeps this coupon (v is the desired destination). Else, v picks a neighbor u uniformly at random and forward C to u.
7: **end for**

Phase 2: Stich short walks by token forwarding. Stitch $\Theta(\ell/\lambda)$ walks, each of length in $[\lambda, 2\lambda - 1]$.

1: The source node s creates a message called "token" which contains the ID of s
2: The algorithm will forward the token around and keep track of a set of *connectors*, denoted by $\mathcal{C}$. Initially, $\mathcal{C} = \{s\}$.
3: **while** Length of walk completed is at most $\ell - 2\lambda$ **do**
4: Let v be the node that is currently holding the token.
5: v calls SAMPLE-COUPON(v) to uniformly sample one of the coupons distributed by v. Let C be the sampled coupon.
6: **if** $v' =$ NULL (all coupons from v have already been deleted) **then**
7: v calls SEND-MORE-COUPONS(v, η, λ) (Perform $\Theta(l/\lambda)$ walks of length $\lambda + r_i$ starting at v, where r_i is chosen uniformly at random in the range $[0, \lambda - 1]$ for the i-th walk.)
8: v calls SAMPLE-COUPON(v) and let C be the returned value
9: **end if**
10: Let v' be node holding coupon C. (ID of v' is written on C.)
11: v sends the token to v' and v' delete C so that C will not be sampled again.
12: $\mathcal{C} = \mathcal{C} \cup \{v'\}$
13: **end while**
14: Walk naively until ℓ steps are completed (this is at most another 2λ steps)
15: A node holding the token outputs the ID of s

Once we have established the correctness, we focus on the running time. In the second part, we show the probabilistic bound of Phase 1.

Lemma 2.3. *Phase 1 finishes in $\tilde{O}(\lambda\eta)$ rounds with high probability.*

In the third part, we analyze the worst case bound of Phase 2, which is a building block of the probabilistic bound.

Lemma 2.4. *Phase 2 finishes in $\tilde{O}(\frac{\ell \cdot D}{\lambda} + \frac{\ell}{\eta})$.*

We note that the above bound holds even when we fix the length of the short walks (instead of randomly picking from $[\lambda, 2\lambda]$. Moreover, using the above lemmas we can conclude the running time of $\tilde{O}(\ell^{2/3}D^{1/3})$ by setting η and λ appropriately.

In the last part, we improve the running time of Phase 2 further, using a probabilistic bound, leading to a better running time overall. The key ingredient here is the *Random Walk Visits Lemma* (cf. Lemma 2.12) stated formally in Section 2.2.4 and proved in Section 2.3. Then we use the fact that the short walks have random length to obtain the running time bound.

Lemma 2.5. *For any η and λ such that $\eta\lambda \geq 32\sqrt{\ell}(\log n)^3$, Phase 2 finishes in $\tilde{O}(\frac{\ell D}{\lambda})$ rounds with high probability.*

Using the results above, we conclude the following theorem.

Theorem 2.6. *For any ℓ, Algorithm* Single-Random-Walk *(cf. Algorithm 2.3) solves 1-RW-DoS correctly and, with high probability, finishes in $\tilde{O}(\sqrt{\ell D})$ rounds.*

Proof. Set $\eta = 1$ and $\lambda = 32\sqrt{\ell D}(\log n)^3$. Using Lemma 2.3 and 2.5, the algorithm finishes in $\tilde{O}(\lambda\eta + \frac{\ell D}{\lambda}) = \tilde{O}(\sqrt{\ell D})$ with high probability. □

2.2.1 Correctness

To prove the correctness of the algorithm, we first claim that Sample-Coupon returns a coupon where the node holding this coupon is a destination of a short walks of length uniformly random in $[\lambda, 2\lambda - 1]$.

Claim 2.7. *Each short walk length (returned by* Sample-Coupon*) is uniformly sampled from the range* $[\lambda, 2\lambda - 1]$.

Proof. Each walk can be created in two ways.

- It is created in Phase 1. In this case, since we pick the length of each walk uniformly from the length $[\lambda, 2\lambda - 1]$, the claim clearly holds.
- It is created by Send-More-Coupon. In this case, the claim holds by the technique of *reservoir sampling*: Observe that after the λ^{th} step of the walk is completed, we stop extending each walk at any length between λ and $2\lambda - 1$ uniformly. To see this, observe that we stop at length λ with probability $1/\lambda$. If the walk does not stop, it will stop at length $\lambda + 1$ with probability $\frac{1}{\lambda-1}$. This means that the walk will stop at length $\lambda + 1$ with probability $\frac{\lambda-1}{\lambda} \times \frac{1}{\lambda-1} = \frac{1}{\lambda}$. Similarly, it can be argue that the walk will stop at length i for any $i \in [\lambda, 2\lambda-1]$ with probability $\frac{1}{\lambda}$.

□

Moreover, we claim that Sample-Coupon(v) samples a short walk uniformly at random among many coupons (and therefore, short walks starting at v).

Claim 2.8. *Algorithm* Sample-Coupon*(v) (cf. Algorithm 2.1), for any node v, samples a coupon distributed by v uniformly at random.*

Proof. Assume that before this algorithm starts, there are t (without loss of generality, let $t > 0$) coupons containing ID of v stored in some nodes in the network. The goal is to show that Sample-Coupon brings one of these coupons to v with uniform probability. For any node u, let T_u be the subtree rooted at u and let S_u be the set of coupons in T_u. (Therefore, $T_v = T$ and $|S_v| = t$.)

We claim that any node u returns a coupon to its parent with uniform probability (i.e., for any coupons $x \in S_u$, $Pr[u \text{ returns } x]$ is $1/|S_u|$ (if $|S_u| > 0$)). We prove this

by induction on the height of the tree. This claim clearly holds for the base case where u is a leaf node. Now, for any non-leaf node u, assume that the claim is true for any of its children. To be precise, suppose that u receives coupons and counts from q children. Assume that it receives coupons $d_1, d_2, ..., d_q$ and counts $c_1, c_2, ..., c_q$ from nodes $u_1, u_2, ..., u_q$, respectively. (Also recall that d_0 is the sample of its own coupons (if exists) and c_0 is the number of its own coupons.) By induction, d_j is sent from u_j to u with probability $1/|S_{u_j}|$, for any $1 \leq j \leq q$. Moreover, $c_j = |S_{u_j}|$ for any j. Therefore, any coupon d_j will be picked with probability $\frac{1}{|S_{u_j}|} \times \frac{c_j}{c_0+c_1+...c_q} = \frac{1}{S_u}$ as claimed.

The lemma follows by applying the claim above to v. □

To conclude, any two $[\lambda, 2\lambda - 1]$-length walk (possibly from different sources) are independent from each other. Moreover, a walk from a particular node is picked uniformly at random. Therefore, algorithm Single-Random-Walk is equivalent to having a source node perform a walk of length between λ and $2\lambda - 1$ and then have the destination do another walk of length between λ and $2\lambda - 1$ and so on.

2.2.2 Analysis of Phase 1

For each coupon C, any $j = 1, 2, ..., \lambda$, and any edge e, we define $X_C^j(e)$ to be a random variable having value 1 if C is sent through e in the j^{th} iteration (i.e., when the counter on C is increased from $j - 1$ to j). Let $X^j(e) = \sum_{C:\text{coupon}} X_C^j(e)$. We compute the expected number of coupons that go through an edge e, as follows.

Claim 2.9. *For any edge e and any j, $\mathbb{E}[X^j(e)] = 2\eta$.*

Proof. Recall that each node v starts with $\eta \deg(v)$ coupons and each coupon takes a random walk. We prove that after any given number of steps j, the expected number of coupons at node v is still $\eta \deg(v)$. Consider the random walk's probability transition matrix, call it A. In this case $Au = u$ for the vector u having value $\frac{\deg(v)}{2m}$ where m is the number of edges in the graph (since this u is the stationary distribution

of an undirected unweighted graph). Now the number of coupons we started with at any node i is proportional to its stationary distribution, therefore, in expectation, the number of coupons at any node remains the same.

To calculate $\mathbb{E}[X^j(e)]$, notice that edge e will receive coupons from its two end points, say x and y. The number of coupons it receives from node x in expectation is exactly the number of coupons at x divided by $\deg(x)$. The claim follows. □

By Chernoff's bound (e.g., in [114, Theorem 4.4.]), for any edge e and any j,

$$\mathbb{P}[X^j(e) \geq 4\eta \log n] \leq 2^{-4 \log n} = n^{-4}.$$

It follows that the probability that there exists an edge e and an integer $1 \leq j \leq \lambda$ such that $X^j(e) \geq 4\eta \log n$ is at most $|E(G)|\lambda n^{-4} \leq \frac{1}{n}$ since $|E(G)| \leq n^2$ and $\lambda \leq \ell \leq n$ (by the way we define λ).

Now suppose that $X^j(e) \leq 4\eta \log n$ for every edge e and every integer $j \leq \lambda$. This implies that we can extend all walks of length i to length $i+1$ in $4\eta \log n$ rounds. Therefore, we obtain walks of length λ in $4\lambda\eta \log n$ rounds as claimed.

2.2.3 Worst-case bound of Phase 2

To prove Lemma 2.4, we first analyze the running time of Send-More-Coupons and Sample-Coupon.

Lemma 2.10. *For any v,* Send-More-Coupons*(v, η, λ) always finishes within $O(\lambda)$ rounds.*

Proof. Consider any node u during the execution of the algorithm. If it contains x coupons of v (i.e., which just contain the ID of v), for some x, it has to pick x of its neighbors at random, and pass the coupon of v to each of these x neighbors. It might pass these coupons to less than x neighbors and cause congestion if the coupons are sent separately. However, it sends only the ID of v and a *count* to each neighbor, where the count represents the number of coupons it wishes to send to such neighbor.

Note that there is only one ID sent during the process since only one node calls Send-More-Coupons at a time. Therefore, there is no congestion and thus the algorithm terminates in $O(\lambda)$ rounds. □

Lemma 2.11. Sample-Coupon *always finishes within* $O(D)$ *rounds.*

Proof. Since, constructing a BFS tree can be done easily in $O(D)$ rounds, it is to bound the time of the second part where the algorithm wishes to *sample* one of many coupons (having its ID) spread across the graph. The sampling is done while retracing the BFS tree starting from leaf nodes, eventually reaching the root. The main observation is that when a node receives multiple samples from its children, it only sends one of them to its parent. Therefore, there is no congestion. The total number of rounds required is therefore the number of levels in the BFS tree, $O(D)$. □

Now we prove the worst-case bound of Phase 2. First, observe that Sample-Coupon is called $O(\frac{\ell}{\lambda})$ times since it is called only by a connector (to fine the next node to forward the token to). By Lemma 2.11, this algorithm takes $O(\frac{\ell \cdot D}{\lambda})$ rounds in total. Next, we claim that Send-More-Coupons is called at most $O(\frac{\ell}{\lambda\eta})$ times in total (summing over all nodes). This is because when a node v calls Send-More-Coupons(v, η, λ), all η walks starting at v must have been stitched and therefore v contributes $\lambda\eta$ steps of walk to the long walk we are constructing. It follows from Lemma 2.10 that Send-More-Coupons algorithm takes $O(\frac{\ell}{\eta})$ rounds in total. The claimed worst-case bound follow by summing up the total running time of both algorithms.

2.2.4 A Probabilistic bound for Phase 2

Recall that we may assume that $\ell = O(m^2)$ (cf. 2.1). We prove the tighter bound using the following lemmas. As mentioned earlier, to bound the number of times Send-More-Coupons is invoked, we need a technical result on random walks that bounds the number of times a node will be visited in a ℓ-length random walk. Consider

a simple random walk on a connected undirected graph on n vertices. Let $d(x)$ denote the degree of x, and let m denote the number of edges. Let $N_t^x(y)$ denote the number of visits to vertex y by time t, given the walk started at vertex x. Now, consider k walks, each of length ℓ, starting from (not necessary distinct) nodes $x_1, x_2, \ldots, x_k$. We show a key technical lemma that applies to a random walk on any graph: With high probability, no vertex y is visited more than $32d(x)\sqrt{k\ell+1}\log n + k$ times.

Lemma 2.12 (Random Walk Visits Lemma)**.** *For any nodes $x_1, x_2, \ldots, x_k$, and $\ell = O(m^2)$,*

$$\Pr\Big(\exists y \ s.t. \ \sum_{i=1}^{k} N_\ell^{x_i}(y) \geq 32d(y)\sqrt{k\ell+1}\log n + k\Big) \leq 1/n\,.$$

Since the proof of this lemma is interesting on its own and lengthly, we defer it to Section 2.3. The lemma above says that the number of visits to each node can be bounded. However, for each node, we are only interested in the case where it is used as a connector. The lemma below shows that the number of visits as a connector can be bounded as well; i.e., if any node v_i appears t times in the walk, then it is likely to appear roughly t/λ times as connectors.

Lemma 2.13. *For any vertex v, if v appears in the walk at most t times then it appears as a connector node at most $t(\log n)^2/\lambda$ times with probability at least $1-1/n^2$.*

At first thought, the lemma above might sound correct even when we do not random the length of the short walks since the connectors are spread out in steps of length approximately λ. However, there might be some *periodicity* that results in the same node being visited multiple times but *exactly* at λ-intervals. This is where we crucially use the fact that the algorithm uses walks of length uniformly random in $[\lambda, 2\lambda-1]$. The proof then goes via constructing another process equivalent to partitioning the ℓ steps into intervals of λ and then sampling points from each interval. We analyze this by carefully constructing a different process that stochastically dominates the process of a node occurring as a connector at various steps in the ℓ-length

walk and then use a Chernoff bound argument.

In order to give a detailed proof of Lemma 2.13, we need the following two claims.

Claim 2.14. *Consider any sequence A of numbers $a_1, ..., a_{\ell'}$ of length ℓ'. For any integer λ', let B be a sequence $a_{\lambda'+r_1}, a_{2\lambda'+r_1+r_2}, ..., a_{i\lambda'+r_1+...+r_i}, ...$ where r_i, for any i, is a random integer picked uniformly from $[0, \lambda' - 1]$. Consider another subsequence of numbers C of A where an element in C is picked from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$. Then, $Pr[C$ contains $a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}]$ for any set $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$.*

Proof. First consider a subsequence C of A. Numbers in C are picked from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$. Observe that $|C| \geq |B|$. In fact, we can say that "C contains B"; i.e., for any sequence of k indexes $i_1, i_2, ..., i_k$ such that $\lambda' \leq i_{j+1} - i_j \leq 2\lambda' - 1$ for all j,

$$Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = Pr[C \text{ contains } \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}].$$

To see this, observe that B will be equal to $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$ only for a specific value of $r_1, r_2, ..., r_k$. Since each of $r_1, r_2, ..., r_k$ is chosen uniformly at random from $[1, \lambda']$, $Pr[B = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = \lambda'^{-k}$. Moreover, the C will contain $a_{i_1}, ai_2, ..., a_{i_k}\}$ if and only if, for each j, we pick a_{i_j} from the interval that contains it (i.e., from $a_{(i'-1)\lambda'+1}, a_{(i'-1)\lambda'+2}, ..., a_{i'\lambda'}$, for some i'). (Note that $a_{i_1}, a_{i_2}, ...$ are all in different intervals because $i_{j+1} - i_j \geq \lambda'$ for all j.) Therefore, $Pr[C$ contains $a_{i_1}, a_{i_2}, ..., a_{i_k}\}] = \lambda'^{-k}$. □

Claim 2.15. *Consider any sequence A of numbers $a_1, ..., a'_\ell$ of length ℓ'. Consider subsequence of numbers C of A where an element in C is picked from from "every λ' numbers" in A; i.e., C consists of $\lfloor \ell'/\lambda' \rfloor$ numbers $c_1, c_2, ...$ where, for any i, c_i is*

chosen uniformly at random from $a_{(i-1)\lambda'+1}, a_{(i-1)\lambda'+2}, ..., a_{i\lambda'}$*. For any number* x*, let* n_x *be the number of appearances of* x *in* A*; i.e.,* $n_x = |\{i \mid a_i = x\}|$*. Then, for any* $R \geq 6n_x/\lambda'$*,* x *appears in* C *more than* R *times with probability at most* 2^{-R}*.*

Proof. For $i = 1, 2, ..., \lfloor \ell'/\lambda' \rfloor$, let X_i be a 0/1 random variable that is 1 if and only if $c_i = x$ and $X = \sum_{i=1}^{\lfloor \ell'/\lambda' \rfloor} X_i$. That is, X is the number of appearances of x in C. Clearly, $E[X] = n_x/\lambda'$. Since X_i's are independent, we can apply the Chernoff bound (e.g., in [114, Theorem 4.4.]): For any $R \geq 6E[X] = 6n_x/\lambda'$,

$$Pr[X \leq R] \geq 2^{-R}.$$

The claim is thus proved. □

Proof of Lemma 2.13. Now we use the claim to prove the lemma. Choose $\ell' = \ell$ and $\lambda' = \lambda$ and consider any node v that appears at most t times. The number of times it appears as a connector node is the number of times it appears in the subsequence B described in the claim. By applying the claim with $R = t(\log n)^2$, we have that v appears in B more than $t(\log n)^2$ times with probability at most $1/n^2$ as desired. □

Now we are ready to prove the probabilistic bound of Phase 2 (cf. Lemma 2.5).

First, we claim, using Lemma 2.12 and 2.13, that each node is used as a connector node at most $\frac{32d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times with probability at least $1-2/n$. To see this, observe that the claim holds if each node x is visited at most $t(x) = 32d(x)\sqrt{\ell+1}\log n$ times and consequently appears as a connector node at most $t(x)(\log n)^2/\lambda$ times. By Lemma 2.12, the first condition holds with probability at least $1 - 1/n$. By Lemma 2.13 and the union bound over all nodes, the second condition holds with probability at least $1 - 1/n$, provided that the first condition holds. Therefore, both conditions hold together with probability at least $1 - 2/n$ as claimed.

Now, observe that Sample-Coupon is invoked $O(\frac{\ell}{\lambda})$ times (only when we stitch the walks) and therefore, by Lemma 2.11, contributes $O(\frac{\ell D}{\lambda}) = \tilde{O}(\sqrt{\ell D})$ rounds.

Moreover, we claim that SEND-MORE-COUPONS is never invoked, with probability at least $1-2/n$. To see this, recall our claim above that each node x is used as a connector node at most $\frac{32d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times. Additionally, observe that we have prepared this many walks in Phase 1; i.e., after Phase 1, each node has $\eta d(x) \geq \frac{32d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ short walks. The claim follows.

Therefore, with probability at least $1 - 2/n$, the rounds are $\tilde{O}(\sqrt{\ell D})$ as claimed.

2.3 Proof of Random Walk Visits Lemma (cf. Lemma 2.12)

We start with the bound of the first and second moment of the number of visits at each node by each walk.

Proposition 2.16. *For any node x, node y and $t = O(m^2)$,*

$$\mathrm{E}[N_t^x(y)] \leq 8d(y)\sqrt{t+1}\,. \tag{1}$$

To prove the above proposition, let P denote the transition probability matrix of such a random walk and let π denote the stationary distribution of the walk, which in this case is simply proportional to the degree of the vertex, and let $\pi_{\min} = \min_x \pi(x)$.

The basic bound we use is the following estimate from Lyons (see Lemma 3.4 and Remark 4 in [111]). Let Q denote the transition probability matrix of a chain with self-loop probablity $\alpha > 0$, and with $c = \min\{\pi(x)Q(x,y) : x \neq y \text{ and } Q(x,y) > 0\}$. Note that for a random walk on an undirected graph, $c = \frac{1}{2m}$. For $k > 0$ a positive integer (denoting time) ,

$$\Big|\frac{Q^k(x,y)}{\pi(y)} - 1\Big| \leq \min\Big\{\frac{1}{\alpha c\sqrt{k+1}}, \frac{1}{2\alpha^2 c^2(k+1)}\Big\}\,. \tag{2}$$

For $k \leq \beta m^2$ for a sufficiently small constant β, and small α, the above can be simplified to the following bound; see Remark 3 in [111].

$$Q^k(x,y) \leq \frac{4\pi(y)}{c\sqrt{k+1}} = \frac{4d(y)}{\sqrt{k+1}}\,. \tag{3}$$

Note that given a simple random walk on a graph G, and a corresponding matrix P, one can always switch to the lazy version $Q = (I + P)/2$, and interpret it as a walk on graph G', obtained by adding self-loops to vertices in G so as to double the degree of each vertex. In the following, with abuse of notation we assume our P is such a lazy version of the original one.

Proof of Proposition 2.16. Let $X_0, X_1, \ldots$ describe the random walk, with X_i denoting the position of the walk at time $i \geq 0$, and let $\mathbf{1}_A$ denote the indicator (0-1) random variable, which takes the value 1 when the event A is true. In the following we also use the subscript x to denote the fact that the probability or expectation is with respect to starting the walk at vertex x. First the expectation.

$$\begin{aligned} \mathrm{E}[N_t^x(y)] &= \mathrm{E}_x[\sum_{i=0}^{t} \mathbf{1}_{\{X_i=y\}}] = \sum_{i=0}^{t} P^i(x,y) \\ &\leq 4d(y) \sum_{i=0}^{t} \frac{1}{\sqrt{i+1}}, \quad \text{(using the above inequality (3))} \\ &\leq 8d(y)\sqrt{t+1}\,. \end{aligned}$$

□

Using the above proposition, we bound the number of visits of each walk at each node, as follows.

Lemma 2.17. *For $t = O(m^2)$ and any vertex $y \in G$, the random walk started at x satisfies:*

$$\Pr\big(N_t^x(y) \geq 32\ d(y)\sqrt{t+1}\log n\big) \leq \frac{1}{n^2}\,.$$

Proof. First, it follows from the Proposition and the Markov's inequality that

$$\Pr\big(N_t^x(y) \geq 4 \cdot 8\ d(y)\sqrt{t+1}\big) \leq \frac{1}{4}\,. \tag{4}$$

For any r, let $L_r^x(y)$ be the time that the random walk (started at x) visits y for the r^{th} time. Observe that, for any r, $N_t^x(y) \geq r$ if and only if $L_r^x(y) \leq t$. Therefore,

$$\Pr(N_t^x(y) \geq r) = \Pr(L_r^x(y) \leq t). \tag{5}$$

Let $r^* = 32\ d(y)\sqrt{t+1}$. By (4) and (5), $\Pr(L^x_{r^*}(y) \le t) \le \frac{1}{4}$. We claim that

$$\Pr(L^x_{r^* \log n}(y) \le t) \le \left(\frac{1}{4}\right)^{\log n} = \frac{1}{n^2}\,. \tag{6}$$

To see this, divide the walk into $\log n$ independent subwalks, each visiting y exactly r^* times. Since the event $L^x_{r^* \log n}(y) \le t$ implies that all subwalks have length at most t, (6) follows. Now, by applying (5) again,

$$\Pr(N^x_t(y) \ge r^* \log n) = \Pr(L^x_{r^* \log n}(y) \le t) \le \frac{1}{n^2}$$

as desired. □

We now extend the above lemma to bound the number of visits of *all* the walks at each particular node.

Lemma 2.18. *For $\gamma > 0$, and $t = O(m^2)$, and for any vertex $y \in G$, the random walk started at x satisfies:*

$$\Pr\Big(\sum_{i=1}^{k} N^{x_i}_t(y) \ge 32\ d(y)\sqrt{kt+1}\log n + k\Big) \le \frac{1}{n^2}\,.$$

Proof. First, observe that, for any r,

$$\Pr\Big(\sum_{i=1}^{k} N^{x_i}_t(y) \ge r - k\Big) \le \Pr[N^y_{kt}(y) \ge r].$$

To see this, we construct a walk W of length kt starting at y in the following way: For each i, denote a walk of length t starting at x_i by W_i. Let τ_i and τ'_i be the first and last time (not later than time t) that W_i visits y. Let W'_i be the subwalk of W_i from time τ_i to τ'_i. We construct a walk W by stitching $W'_1, W'_2, ..., W'_k$ together and complete the rest of the walk (to reach the length kt) by a normal random walk. It then follows that the number of visits to y by $W_1, W_2, \ldots, W_k$ (excluding the starting step) is at most the number of visits to y by W. The first quantity is $\sum_{i=1}^{k} N^{x_i}_t(y) - k$. (The term '$-k$' comes from the fact that we do not count the first visit to y by each

W_i which is the starting step of each W'_i.) The second quantity is $N^y_{kt}(y)$. The observation thus follows.

Therefore,

$$\Pr\Big(\sum_{i=1}^{k} N^{x_i}_t(y) \geq 32\, d(y)\sqrt{kt+1}\log n + k\Big) \leq \Pr\Big(N^y_{kt}(y) \geq 32\, d(y)\sqrt{kt+1}\log n\Big) \leq \frac{1}{n^2}$$

where the last inequality follows from Lemma 2.17. □

The Random Walk Visits Lemma (cf. Lemma 2.12) follows immediately from Lemma 2.18 by union bounding over all nodes.

2.4 Variations, Extensions, and Generalizations

2.4.1 Computing k Random Walks

We now consider the scenario when we want to compute k walks of length ℓ from different (not necessary distinct) sources $s_1, s_2, \ldots, s_k$. We show that Single-Random-Walk can be extended to solve this problem. Consider the following algorithm.

Many-Random-Walks**:** Let $\lambda = (32\sqrt{k\ell D+1}\log n + k)(\log n)^2$ and $\eta = 1$. If $\lambda > \ell$ then run the naive random walk algorithm, i.e., the sources find walks of length ℓ simultaneously by sending tokens. Otherwise, do the following. First, modify Phase 2 of Single-Random-Walk to create multiple walks, one at a time; i.e., in the second phase, we stitch the short walks together to get a walk of length ℓ starting at s_1 then do the same thing for s_2, s_3, and so on.

Theorem 2.19. Many-Random-Walks *finishes in* $\tilde{O}\left(\min(\sqrt{k\ell D}+k, k+\ell)\right)$ *rounds with high probability.*

Proof. First, consider the case where $\lambda > \ell$. In this case, $\min(\sqrt{k\ell D}+k, \sqrt{k\ell}+k+\ell) = \tilde{O}(\sqrt{k\ell}+k+\ell)$. By Lemma 2.12, each node x will be visited at most $\tilde{O}(d(x)(\sqrt{k\ell}+k))$ times. Therefore, using the same argument as Lemma 2.3, the

congestion is $\tilde{O}(\sqrt{k\ell} + k)$ with high probability. Since the dilation is ℓ, MANY-RANDOM-WALKS takes $\tilde{O}(\sqrt{k\ell} + k + \ell)$ rounds as claimed. Since $2\sqrt{k\ell} \leq k + \ell$, this bound reduces to $O(k + \ell)$.

Now, consider the other case where $\lambda \leq \ell$. In this case, $\min(\sqrt{k\ell D} + k, \sqrt{k\ell} + k + \ell) = \tilde{O}(\sqrt{k\ell D} + k)$. Phase 1 takes $\tilde{O}(\lambda\eta) = \tilde{O}(\sqrt{k\ell D} + k)$. The stitching in Phase 2 takes $\tilde{O}(k\ell D/\lambda) = \tilde{O}(\sqrt{k\ell D})$. Moreover, by Lemma 2.12, SEND-MORE-COUPONS will never be invoked. Therefore, the total number of rounds is $\tilde{O}(\sqrt{k\ell D} + k)$ as claimed. □

2.4.2 Regenerating the entire random walk

Our algorithm can be extended to regenerate the entire walk, solving k-RW-pos. This will be use, e.g., in generating a random spanning tree. The algorithm is the following. First, inform all intermediate connecting nodes of their position which can be done by keeping trick of the walk length when we do token forwarding in Phase 2. Then, these nodes can regenerate their $O(\sqrt{\ell})$ length short walks by simply sending a message through each of the corresponding short walks. This can be completed in $\tilde{O}(\sqrt{\ell D})$ rounds with high probability. This is because, with high probability, SEND-MORE-COUPONS will not be invoked and hence all the short walks are generated in Phase 1. Sending a message through each of these short walks (in fact, sending a message through *every* short walk generated in Phase 1) takes time at most the time taken in Phase 1, i.e., $\tilde{O}(\sqrt{\ell D})$ rounds.

2.4.3 Generalization to the Metropolis-Hastings algorithm

We now discuss extensions of our algorithm to perform random walk according to the Metropolis-Hastings algorithm, a more general type of random walk with numerous applications (e.g., [155]). The Metropolis-Hastings [75, 113] algorithm gives a way to define a transition probability so that a random walk converges to any desired distribution π (where π_i, for any node i, is the desired stationary distribution at node

i). It is assumed that every node i knows its steady state distribution π_i (and can know its neighbors' steady state distribution in one round).

The algorithm is roughly as follows (see, e.g., [75, 113] for full descriptions). For any desired distribution π and any desired *laziness factor* $0 < \alpha < 1$, the transition probability from node i to its neighbor j is defined to be

$$P_{ij} = \alpha \min(1/d_i, \pi_j/(\pi_i d_j))$$

where d_i and d_j are degree of i and j respectively. It is shown that a random walk with this transition probability converges to π.

Using the transition probability defined above, we now run the Single-Random-Walk algorithm with one modification: in Phase 1, we generate

$$\eta \cdot \frac{\pi(x)}{\alpha \min_x \frac{\pi(x)}{d(x)}}$$

short walks instead of $\eta d(v)$ walks. The correctness of the algorithm follows from Lemma 2.2. The running time follows from the following theorem.

Theorem 2.20. *For any η and λ such that $\eta\lambda \geq 32\sqrt{\ell D}(\log n)^3$, the modified* Single-Random-Walk *algorithm stated above finishes in*

$$\tilde{O}(\lambda\eta \log n \cdot \frac{\max_x \pi(x)/d(x)}{\min_y \pi(y)/d(y)} + \frac{\ell D}{\lambda})$$

rounds with high probability.

Like Theorem 2.6, the above theorem follows from the following two lemmas which are similar to Lemma 2.3 and 2.5.

Lemma 2.21. *For any π and α, Phase 1 finishes in $O(\lambda\eta \log n \cdot \frac{\max_x \pi(x)/d(x)}{\min_y \pi(y)/d(y)})$ rounds with high probability.*

Proof. The proof is essentially the same as Lemma 2.3. We present it here for completeness. Let $\beta = \frac{1}{\alpha \min_x \frac{\pi(x)}{d(x)}}$. Consider the case when each node i creates $\beta\pi(i)\eta$ messages. We show that the lemma holds even in this case.

We use the same definition as in Lemma 2.3. That is, for each message M, any $j = 1, 2, ..., \lambda$, and any edge e, we define $X_M^j(e)$ to be a random variable having value 1 if M is sent through e in the j^{th} iteration (i.e., when the counter on M has value $j-1$). Let $X^j(e) = \sum_{M:\text{message}} X_M^j(e)$. We compute the expected number of messages that go through an edge. As before, we show the following claim.

Claim 2.22. *For any edge e and any j, $\mathbb{E}[X^j(e)] = 2\eta \cdot \frac{\max_x \pi(x)/d(x)}{\min_y \pi(y)/d(y)}$.*

Proof. Assume that each node v starts with $\beta\pi(v)\eta$ messages. Each message takes a random walk. We prove that after any given number of steps j, the expected number of messages at node v is still $\beta\pi(v)\eta$. Consider the random walk's probability transition matrix, say A. In this case $Au = u$ for the vector u having value $\pi(v)$ (since this $\pi(v)$ is the stationary distribution). Now the number of messages we started with at any node i is proportional to its stationary distribution, therefore, in expectation, the number of messages at any node remains the same.

To calculate $\mathbb{E}[X^j(e)]$, notice that edge e will receive messages from its two end points, say x and y. The number of messages it receives from node x in expectation is exactly $\beta\pi(x)\eta \times \alpha \min(\frac{1}{d_x}, \frac{\pi_y}{\pi_x d_y}) \leq \eta \cdot \frac{\pi(x)/d(x)}{\min_y \pi(y)/d(y)}$. The claim follows. □

The rest analysis follows the same way as the analysis of Lemma 2.3. □

Lemma 2.23. *For any η and λ such that $\eta\lambda \geq 32\sqrt{\ell}(\log n)^3$, Phase 2 finishes in $\tilde{O}(\frac{\ell D}{\lambda})$ rounds with high probability.*

Proof. (Sketched) We first prove a result similar to Proposition 2.16

Claim 2.24. *For any node x, node y and $t = O(m^2)$,*

$$\mathrm{E}[N_t^x(y)] \leq \frac{8\pi(y)\sqrt{t+1}}{\alpha \min_x \pi(x)/d(x)}\,. \tag{7}$$

Proof. The proof is similar to the proof of Lemma 2.16 except that

$$c = \alpha \min_x \pi(x)/d(x).$$

It follows that

$$\begin{aligned}
\mathrm{E}[N_t^x(y)] &= \mathrm{E}_x[\sum_{i=0}^{t} \mathbf{1}_{\{X_i=y\}}] = \sum_{i=0}^{t} P^i(x,y) \\
&\leq \frac{4\pi(y)}{c} \sum_{i=0}^{t} \frac{1}{\sqrt{i+1}}, \quad \text{(using the above inequality (3))} \\
&\leq \frac{8\pi(y)\sqrt{t+1}}{\alpha \min_x \pi(x)/d(x)}\,.
\end{aligned}$$

□

By following the rest of the proof of Lemma 2.12, we conclude the following.

Claim 2.25. *For any nodes $x_1, x_2, \ldots, x_k$, and $\ell = O(m^2)$,*

$$\Pr\big(\exists y \ s.t. \ \sum_{i=1}^{k} N_\ell^{x_i}(y) \geq 32 \frac{\pi(y)}{\alpha \min_x \pi(x)/d(x)} \sqrt{k\ell+1} \log n + k\big) \leq 1/n\,.$$

Following the proof of Lemma 2.5, we have that each node y is used as a connector at most

$$\frac{32(\frac{\pi(y)}{\alpha \min_x \pi(x)/d(x)})\sqrt{\ell}(\log n)^3}{\lambda}$$

times with probability at least $1 - 2/n$. Additionally, observe that we have prepared this many walks in Phase 1; i.e., after Phase 1, each node x has

$$\eta \cdot \frac{\pi(x)}{\alpha \min_x \frac{\pi(x)}{d(x)}} \geq \frac{32(\frac{\pi(x)}{\alpha \min_y \pi(y)/d(y)})\sqrt{\ell}(\log n)^3}{\lambda}$$

short walks. The claim follows. □

An interesting application of the above theorem is when π is a uniform distribution. In this case, we can compute a random walk of length ℓ in $\tilde{O}(\lambda\eta \log n \cdot \frac{\max_y d(y)}{\min_x d(x)} + \frac{\ell D}{\lambda})$.

2.5 Applications

In this section, we present two applications of our algorithm.

2.5.1 A Distributed Algorithm for Random Spanning Tree

We now present an algorithm for generating a random spanning tree (RST) of an unweighted undirected network in $\tilde{O}(\sqrt{mD})$ rounds with high probability. The approach is to simulate Aldous and Broder's [5, 22] RST algorithm which is as follows. First, pick one arbitrary node as a root. Then, perform a random walk from the root node until all nodes are visited. For each non-root node, output the edge that is used for its first visit. (That is, for each non-root node v, if the first time v is visited is t then we output the edge (u, v) where u is the node visited at time $t-1$.) The output edges clearly form a spanning tree and this spanning tree is shown to come from a uniform distribution among all spanning trees of the graph [5, 22]. The running time of this algorithm is bounded by the time to visit all the nodes of the the graph which can shown to be $\tilde{O}(mD)$ (in the worst case, i.e., for any undirected, unweighted graph) by Aleniunas et al. [6].

This algorithm can be simulated on the distributed network by our random walk algorithm as follows. The algorithm can be viewed in phases. Initially, we pick a root node arbitrarily and set $\ell = n$. In each phase, we run $\log n$ (different) walks of length ℓ starting from the root node (this takes $\tilde{O}(\sqrt{\ell D})$ rounds using our distributed random walk algorithm). If none of the $O(\log n)$ different walks cover all nodes (this can be easily checked in $O(D)$ time), we double the value of ℓ and start a new phase, i.e., perform again $\log n$ walks of length ℓ. The algorithm continues until one walk of length ℓ covers all nodes. We then use such walk to construct a random spanning tree: As the result of this walk, each node knows its position(s) in the walk (cf. Section 2.2), i.e., it has a list of steps in the walk that it is visited. Therefore, each non-root node can pick an edge that is used in its first visit by communicating to its neighbors. Thus at the end of the algorithm, each node can know which of its adjacent edges belong to the output tree. (An additional $O(n)$ rounds may be used to deliver the resulting tree to a particular node if needed.)

We now analyze the number of rounds in term of τ, the expected cover time of the input graph. The algorithm takes $O(\log \tau)$ phases before $2\tau \leq \ell \leq 4\tau$, and since one of $\log n$ random walks of length 2τ will cover the input graph with high probability, the algorithm will stop with $\ell \leq 4\tau$ with high probability. Since each phase takes $\tilde{O}(\sqrt{\ell D})$ rounds, the total number of rounds is $\tilde{O}(\sqrt{\tau D})$ with high probability. Since $\tau = \tilde{O}(mD)$, we have the following theorem.

Theorem 2.26. *The algorithm described above generates a uniform random spanning tree in $\tilde{O}(\sqrt{m}D)$ rounds with high probability.*

2.5.2 Decentralized Estimation of Mixing Time

We now present an algorithm to estimate the mixing time of a graph from a specified source. Throughout this section, we assume that the graph is connected and non-bipartite (the conditions under which mixing time is well-defined). The main idea in estimating the mixing time is, given a source node, to run many random walks of length ℓ using the approach described in the previous section, and use these to estimate the distribution induced by the ℓ-length random walk. We then compare the distribution at length ℓ, with the stationary distribution to determine if they are *close*, and if not, double ℓ and retry. For this approach, one issue that we need to address is how to compare two distributions with few samples efficiently (a well-studied problem). We introduce some definitions before formalizing our approach and theorem.

Definition 2.27 (Distribution vector)**.** Let $\pi_x(t)$ define the probability distribution vector reached after t steps when the initial distribution starts with probability 1 at node x. Let π denote the stationary distribution vector.

Definition 2.28 ($\tau^x(\epsilon)$ and τ^x_{mix}, mixing time for source x)**.** Define $\tau^x(\epsilon) = \min t : ||\pi_x(t) - \pi||_1 < \epsilon$. Define $\tau^x_{mix} = \tau^x(1/2e)$.

The goal is to estimate τ^x_{mix}. Notice that the definition of τ^x_{mix} is consistent due to the following standard monotonicity property of distributions (proof in the appendix).

Lemma 2.29. $||\pi_x(t+1) - \pi||_1 \leq ||\pi_x(t) - \pi||_1$.

Proof. The monotonicity follows from the fact that $||Ax||_1 \leq ||x||_1$ where A is the transpose of the transition probability matrix of the graph and x is any probability vector. That is, $A(i,j)$ denotes the probability of transitioning from node j to node i. This in turn follows from the fact that the sum of entries of any column of A is 1.

Now let π be the stationary distribution of the transition matrix A. This implies that if ℓ is ϵ-near mixing, then $||A^l u - \pi||_1 \leq \epsilon$, by definition of ϵ-near mixing time. Now consider $||A^{l+1}u - \pi||_1$. This is equal to $||A^{l+1}u - A\pi||_1$ since $A\pi = \pi$. However, this reduces to $||A(A^l u - \pi)||_1 \leq \epsilon$. It follows that $(\ell + 1)$ is ϵ-near mixing. □

To compare two distributions, we use the technique of Batu et. al. [18] to determine if the distributions are ϵ-near. Their result (slightly restated) is summarized in the following theorem.

Theorem 2.30 ([18])**.** *For any ϵ, given $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples of a distribution X over $[n]$, and a specified distribution Y, there is a test that outputs PASS with high probability if $|X - Y|_1 \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, and outputs FAIL with high probability if $|X - Y|_1 \geq 6\epsilon$.*

We now give a very brief description of the algorithm of Batu et. al. [18] to illustrate that it can in fact be simulated on the distributed network efficiently. The algorithm partitions the set of nodes in to buckets based on the steady state probabilities. Each of the $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from X now falls in one of these buckets. Further, the actual count of number of nodes in these buckets for distribution Y are counted. The exact count for Y for at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets (corresponding to the samples) is compared with the number of samples from X; these are compared

to determine if X and Y are close. We refer the reader to their paper [18] for a precise description.

Our algorithm starts with $\ell = 1$ and runs $K = \tilde{O}(\sqrt{n})$ walks of length ℓ from the specified source x. As the test of comparison with the steady state distribution outputs FAIL (for choice of $\epsilon = 1/12e$), ℓ is doubled. This process is repeated to identify the largest ℓ such that the test outputs FAIL with high probability and the smallest ℓ such that the test outputs PASS with high probability. These give lower and upper bounds on the required τ^x_{mix} respectively. Our resulting theorem is presented below.

Theorem 2.31. *Given a graph with diameter D, a node x can find, in $\tilde{O}(n^{1/2} + n^{1/4}\sqrt{D\tau^x(\epsilon)})$ rounds, a time $\tilde{\tau}^x_{mix}$ such that $\tau^x_{mix} \leq \tilde{\tau}^x_{mix} \leq \tau^x(\epsilon)$, where $\epsilon = \frac{1}{6912e\sqrt{n}\log n}$.*

Proof. For undirected unweighted graphs, the stationary distribution of the random walk is known and is $\frac{deg(i)}{2m}$ for node i with degree $deg(i)$, where m is the number of edges in the graph. If a source node in the network knows the degree distribution, we only need $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from a distribution to compare it to the stationary distribution. This can be achieved by running MultipleRandomWalk to obtain $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ random walks. We choose $\epsilon = 1/12e$. To find the approximate mixing time, we try out increasing values of l that are powers of 2. Once we find the right consecutive powers of 2, the monotonicity property admits a binary search to determine the exact value for the specified ϵ.

The result in [18] can also be adapted to compare with the steady state distribution even if the source does not know the entire distribution. As described previously, the source only needs to know the *count* of number of nodes with steady state distribution in given buckets. Specifically, the buckets of interest are at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ as the count is required only for buckets were a sample is drawn from. Since each node knows its own steady state probability (determined just by its degree), the source can broadcast a specific bucket information and recover, in $O(D)$ steps, the count of

number of nodes that fall into this bucket. Using upcast, the source can obtain the bucket count for each of these at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets in $\tilde{O}(n^{1/2}poly(\epsilon^{-1}) + D)$ rounds.

By Theorem 2.19, a source node can obtain K samples from K independent random walks of length ℓ in $\tilde{O}(K+\sqrt{KlD})$ rounds. Setting $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1})+D)$ completes the proof. □

Suppose our estimate of τ^x_{mix} is close to the mixing time of the graph defined as $\tau_{mix} = \max_x \tau^x_{mix}$, then this would allow us to estimate several related quantities. Given a mixing time τ_{mix}, we can approximate the spectral gap $(1 - \lambda_2)$ and the conductance (Φ) due to the known relations that $\frac{1}{1-\lambda_2} \leq \tau_{mix} \leq \frac{\log n}{1-\lambda_2}$ and $\Theta(1-\lambda_2) \leq \Phi \leq \Theta(\sqrt{1-\lambda_2})$ as shown in [79].

2.6 Conclusions

This chapter gives a tight upper bound on the time complexity of distributed computation of random walks in undirected networks. The running time of our algorithm is optimal matching the lower bound that will be shown in Chapter 4. However, our upper bound for performing k independent random walks may not be tight and it will be interesting to resolve this. While the focus of this chapter is on time complexity, message complexity is also important. While our algorithm has a good *amortized* message complexity over several random walks, for doing one walk our algorithm takes much more messages than the naive token passing algorithm that takes ℓ messages. It would be nice to come up with an algorithm that is round efficient and yet has smaller message complexity.

We presented two algorithmic applications of our distributed random walk algorithm: estimating mixing times and computing random spanning trees. It would be interesting to improve upon these results. For example, is there a $\tilde{O}(\sqrt{\tau^x_{mix}} + n^{1/4})$

round algorithm to estimate τ^x; and is there an algorithm for estimating the mixing time (which is the worst among all starting points)? Another open question is whether there exists a $\tilde{O}(n)$ round (or a faster) algorithm for RST?

There are several interesting directions to take this work further. Can these techniques be useful for estimating the second eigenvector of the transition matrix (useful for sparse cuts)? Are there efficient distributed algorithms for random walks in directed graphs (useful for PageRank and related quantities)? Finally, from a practical standpoint, it is important to develop algorithms that are robust to failures and it would be nice to extend our techniques to handle such node/edge failures. This can be useful for doing decentralized computation in large-scale dynamic networks.

Related publications. The preliminary versions of this chapter appeared as joint results with Atish Das Sarma, Gopal Pandurangan, and Prasad Tetali [44, 45].

CHAPTER III

FROM COMMUNICATION COMPLEXITY TO DISTRIBUTED ALGORITHM LOWER BOUNDS

In this chapter, we show a connection between communication complexity and distributed algorithm lower bounds. In particular, we show through a fairly simple reduction that communication complexity lower bounds can be used to prove distributed algorithm lower bounds. Surprisingly, this simple connection proves to have many applications, as we will show later in Chapter 4 and 5.

This chapter is organized as follows. In Section 3.1, we define the model called *distributed communication complexity* which could be thought of as variations of both distributed networks and communication complexity. We also define some variants of the communication complexity model. The main theorems that will be used later in Chapter 4 and 5 can be found in this section as well. In Section 3.2, we define two types of distributed networks that we are particularly interested in. We then prove the main theorems in the last two sections of this chapter.

3.1 Two-party distributed communication complexity

Recall the following basic communication complexity problem. There are two parties that have unbounded computational power. Each party receives a b-bit string, for some integer $b \geq 1$, denoted by $x, y \in \{0,1\}^b$. They both want to together compute $f(x,y)$ for some function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$. In this chapter, we consider three versions of this problem.

- *Direct communication:* This is the standard model in communication complexity introduced in Section 1.1. Two parties can communicate via a bidirectional

edge of unlimited bandwidth. We call the party receiving x *Alice*, and the other party *Bob*. At the end of the process, Bob will output $f(x, y)$. The goal of this model is to minimize the number of bits communicated.

- *r-round direct communication:* This is a variant of the above communication complexity model (see [121] and references therein for papers that considered this model). Like before, Alice and Bob can communicate via a bidirectional edge of unlimited bandwidth. However, two parties are allowed to communicate for only r *rounds* where in each round Alice sends a message (of any size) to Bob followed by Bob sending a message to Alice.

- *Distributed communication on network G:* Two parties are distinct nodes in some B-model network G. We denote the nodes receiving x and y by s and t, respectively. At the end of the process, r will output $f(x, y)$. The goal of this model is to minimize time (i.e., the number of rounds).

We consider time lower bounds for *public coin randomized algorithms* under all models. In particular, we assume that all parties (Alice and Bob in the first two model and all nodes in G in the last model) share a random bit string of infinite length.

Recall that, for any $\epsilon \geq 0$, we say that a randomized algorithm $\mathcal{A}$ is *ϵ-error* if for any input, it outputs the correct answer with probability at least $1 - \epsilon$, where the probability is over all possible random bit strings. The running time of $\mathcal{A}$, denoted by $T_{\mathcal{A}}$, is the number of rounds in the worst case (over all inputs and random strings).

In the first two models, we focus on the message complexity, i.e., the total number of bits exchanged between Alice and Bob, denoted by $R_{\epsilon}^{cc-pub}(f)$ for the first model and $R_{\epsilon}^{r-cc-pub}(f)$ for the second model. In the last model, we focus on the running time, denoted by $R_{\epsilon}^{G,s,t}(f)$.

Networks $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$. Two networks that we are particularly interested in are $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$, for some integers Γ, Λ, p, and d, and real κ. The network $G_1(\Gamma, p, d)$ was introduced in [55] (which was built upon [129, 107]) while $G_2(\Gamma, \Lambda, \kappa)$ is newly constructed in this thesis based on the networks considered in [129, 107].

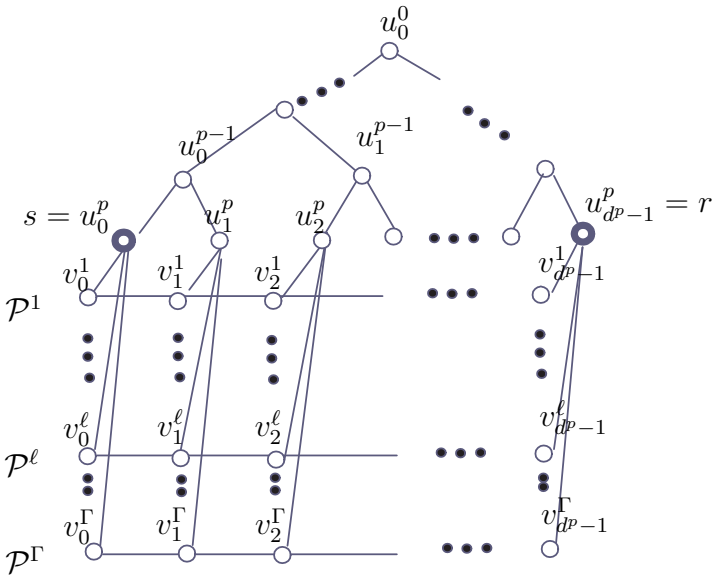
Before we describe these networks in detail, we note the following characteristics which will be used in later sections. An essential parts of both networks consists of Γ *paths*, denoted by $\mathcal{P}^1, \ldots, \mathcal{P}^\Gamma$ and nodes s and t (see Figure 7). Every edge induced by this subgraph has infinitely many copies (in other words, infinite capacity). (We let some edges to have infinitely many copies so that we will have a freedom to specify the number of copies later on when we prove Theorem 4.1 in Section 4.2. The leftmost and rightmost nodes of each path are adjacent to s and t respectively. Ending nodes on the same side of the path (i.e., leftmost or rightmost nodes) are adjacent to each other.

The lemmas below state the important properties of both networks. The first lemma is proved in [55] while the second will be proved after we describe it in detail in Section 3.2.3.

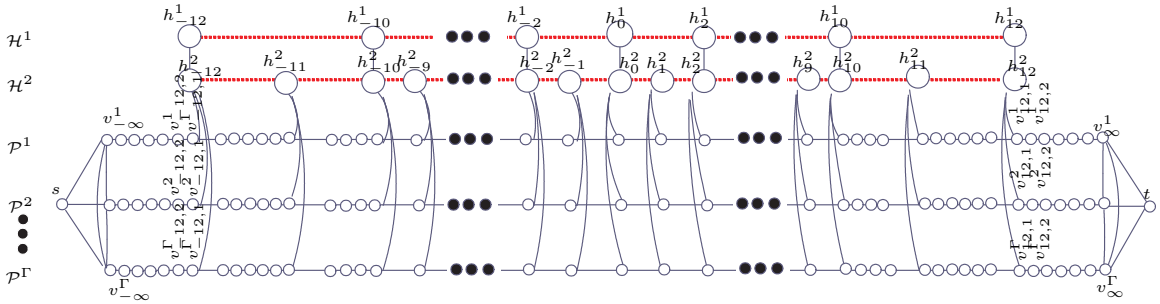
Lemma 3.1. *[55] The number of nodes in $G_1(\Gamma, p, d)$ is $n = \Theta(\Gamma d^p)$ nodes. Each of its path $\mathcal{P}^i$ has d^p nodes. Its diameter is $D = 2p + 2$.*

Lemma 3.2. *For any $\Gamma \geq 1$, $\kappa \geq 1$ and $\Lambda \geq 2$, network $G_2(\Gamma, \Lambda, \kappa)$ has $n = \Theta(\Gamma\kappa\Lambda^\kappa)$ nodes. Each of its path $\mathcal{P}^i$ has $\Theta(\kappa\Lambda^\kappa)$ nodes. Its diameter is $D = \Theta(\kappa\Lambda)$.*

We note that each of $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$ has some properties that the other network does not have. For example, we can construct a network of diameter four using $G_1(\Gamma, p, d)$ but not $G_2(\Gamma, \Lambda, \kappa)$; on the other hand, we can construct a network of diameter $\Omega(n)$ from $G_2(\Gamma, \Lambda, \kappa)$ but not $G_1(\Gamma, p, d)$. Network $G_1(\Gamma, p, d)$ is useful when we want to show a lower bound on networks of low diameter (e.g., the lower



(a) Example of $G_1(\Gamma, p, d)$ (here $d = 2$)



(b) Example of $G_2(\Gamma, \Lambda, \kappa)$ (here, $\kappa = 2.5$ and $\Lambda = 2$)

Figure 7: An example of $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$. The dashed edges (in red) have one copy while other edges have infinitely many copies.

bound of $\Omega(\sqrt{n})$ on networks of diameter $O(\log n)$ shown in Chapter 5). On the other hand, network $G_2(\Gamma, \Lambda, \kappa)$ is useful when we want a large network diameter to play a role in the lower bound (e.g., the lower bound of $\Omega(\sqrt{\ell D})$ on networks of diameter D shown in Chapter 4.)

Main theorems. In this chapter we prove the following theorems. The first theorem states that if there is a fast ϵ-error algorithm for computing f on $G_1(\Gamma, p, d)$, then there is an efficient ϵ-error algorithm for Alice and Bob to compute f. The second theorem says that if there is a fast ϵ-error algorithm for computing function

f on $G_2(\Gamma, \Lambda, \kappa)$, then there is an efficient bounded-round ϵ-error algorithm for Alice and Bob to compute f; moreover, the number of rounds depends on the diameter of $G_2(\Gamma, \Lambda, \kappa)$ (which is $\Theta(\kappa\Lambda)$).

Theorem 3.3. *For any Γ, d, p, B, $\epsilon \geq 0$, and function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$, if*

$$R_\epsilon^{G_1(\Gamma,p,d),s,r}(f) < \frac{d^p - 1}{2}$$

then f can be computed by a direct communication protocol using at most $2dpBR_\epsilon^{G_1(\Gamma,p,d),s,r}(f)$ communication bits in total. In other words,

$$R_\epsilon^{cc-pub}(f) \leq 2dpBR_\epsilon^{G_1(\Gamma,p,d),s,r}(f)\,.$$

Theorem 3.4. *For any $\Gamma \geq 1$, $\Lambda \geq 2$, $\kappa \geq 1$, B, $\epsilon \geq 0$ and function $f : \{0,1\}^b \times \{0,1\}^b \to \mathbb{R}$, for any b, if*

$$R_\epsilon^{G_2(\Gamma,\Lambda,\kappa),s,t}(f) \leq \kappa\Lambda^\kappa$$

then f can be computed by a $\frac{8R_\epsilon^{G_2(\Gamma,\Lambda,\kappa),s,t}(f)}{\kappa\Lambda}$-round direct communication protocol using at most $2\kappa BR_\epsilon^{G_2(\Gamma,\Lambda,\kappa),s,t}(f)$ communication bits in total. In other words,

$$R_\epsilon^{\frac{8R_\epsilon^{G_2(\Gamma,\Lambda,\kappa),s,t}(f)}{\kappa\Lambda}-cc-pub}(f) \leq 2\kappa BR_\epsilon^{G_2(\Gamma,\Lambda,\kappa),s,t}(f)\,.$$

3.2 Descriptions of $G_1(\Gamma, p, d)$ and $G_2(\Gamma, \Lambda, \kappa)$

3.2.1 Description of $G_1(\Gamma, p, d)$

The two basic units in the construction are *paths* and a *tree*. There are Γ paths, denoted by $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^\Gamma$, each having d^p nodes, i.e., for $\ell = 1, 2, \ldots \Gamma$,

$$V(\mathcal{P}^\ell) = \{v_0^\ell, \ldots, v_{d^p-1}^\ell\} \qquad \text{and} \qquad E(\mathcal{P}^\ell) = \{(v_i^\ell, v_{i+1}^\ell) \mid 0 \leq i < d^p - 1\}\,.$$

There is a tree, denoted by $\mathcal{T}$ having depth p where each non-leaf node has d children (thus, there are d^p leaf nodes). We denote the nodes of $\mathcal{T}$ at level ℓ from left to right by $u_0^\ell, \ldots, u_{d^\ell-1}^\ell$ (so, u_0^0 is the root of $\mathcal{T}$ and $u_0^p, \ldots, u_{d^p-1}^p$ are the leaves of $\mathcal{T}$). For any ℓ and j, the leaf node u_j^p is connected to the corresponding path node v_j^ℓ by a

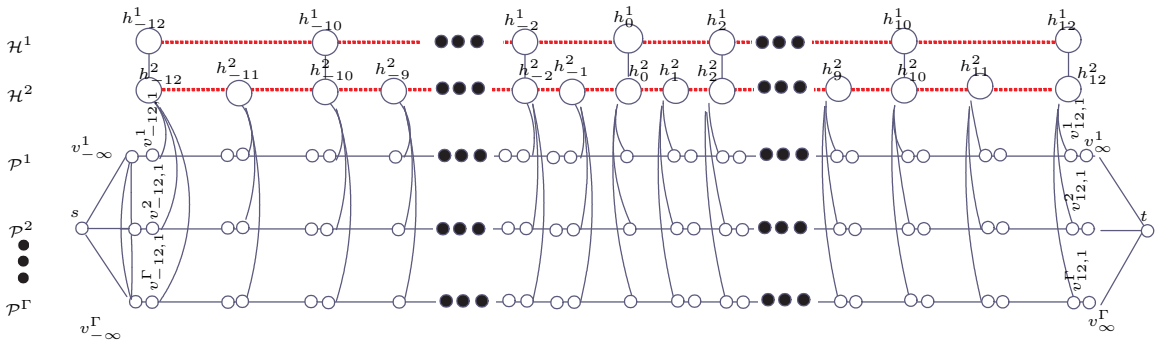


Figure 8: An example of $F(\Gamma, \kappa, \Lambda)$ where $\Lambda = 2$ and $2 \leq \kappa < 3$.

spoke edge (u_j^p, v_j^ℓ). We set the two special nodes (which will receive input strings x and y) as $s = u_0^p$ and $r = u_{d^p-1}^p$. Finally, we create infinitely many copies of all edges except those in $\mathcal{T}$. Figure 7(a) depicts this network.

3.2.2 Preliminary: the network $F(\Gamma, \kappa, \Lambda)$

Before we describe the construction of $G_2(\Gamma, \Lambda, \kappa)$, we first describe a network called $F(\Gamma, \kappa, \Lambda)$ which is a slight modification of the network F_m^K introduced in [129]. In the next section, we show how we modify $F(\Gamma, \kappa, \Lambda)$ to obtain $G_2(\Gamma, \Lambda, \kappa)$.

$G_2(\Gamma, \Lambda, \kappa)$ has three parameters, a real $\kappa \geq 1$ and two integers $\Gamma \geq 1$ and $\Lambda \geq 2$.[1] The two basic units in the construction of $F(\Gamma, \kappa, \Lambda)$ are *highways* and *paths*.

Highways. There are $\lfloor \kappa \rfloor$ highways, denoted by $\mathcal{H}^1$, $\mathcal{H}^2$, ..., $\mathcal{H}^{\lfloor \kappa \rfloor}$. The highway $\mathcal{H}^i$ is a path of $2\lceil \kappa \rceil \Lambda^i + 1$ nodes, i.e.,

$$V(\mathcal{H}^i) = \{h_0^i, h_{\pm\Lambda^{\lfloor \kappa \rfloor - i}}^i, h_{\pm 2\Lambda^{\lfloor \kappa \rfloor - i}}^i, h_{\pm 3\Lambda^{\lfloor \kappa \rfloor - i}}^i, \ldots, h_{\pm \lceil \kappa \rceil \Lambda^i \Lambda^{\lfloor \kappa \rfloor - i}}^i\}$$

$$E(\mathcal{H}^i) = \{(h_{-(j+1)\Lambda^{\lfloor \kappa \rfloor - i}}^i, h_{-j\Lambda^{\lfloor \kappa \rfloor - i}}^i), (h_{j\Lambda^{\lfloor \kappa \rfloor - i}}^i, h_{(j+1)\Lambda^{\lfloor \kappa \rfloor - i}}^i) \mid 0 \leq j < \lceil \kappa \rceil \Lambda^i\}.$$

We connect the highways by adding edges between nodes of the same subscripts, i.e., for any $0 < i \leq \lfloor \kappa \rfloor$ and $-\lceil \kappa \rceil \Lambda^i \leq j \leq \lceil \kappa \rceil \Lambda^i$, there is an edge between $h_{j\Lambda^{\lfloor \kappa \rfloor - i}}^i$

[1] Note that we could restrict κ to be an integer here since $F(\Gamma, \kappa, \Lambda) = F(\Gamma, \kappa', \Lambda)$ for any Λ, Γ, κ and κ' such that $\lfloor \kappa \rfloor = \lfloor \kappa' \rfloor$. However, we will need κ to be a real when we define $G_2(\Gamma, \Lambda, \kappa)$ so we allow it to be a real here as well to avoid confusion.

and $h^{i+1}_{j\Lambda^{\lfloor\kappa\rfloor}-i}$.

For any $j \neq 0$, let

$$\phi_j = 1 \text{ if } j = 0, \text{ and } \phi'_j = \Lambda \text{ otherwise.} \tag{8}$$

We use ϕ'_j to specify the number of nodes in the paths (defined next), i.e., each path will have $\sum_{j=-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_j$ nodes. Note that

$$\sum_{j=-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_j = (2\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}+1)\Lambda = \Theta(\kappa\Lambda^{\lfloor\kappa\rfloor+1}). \tag{9}$$

Paths. There are Γ paths, denoted by $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^\Gamma$. To construct each path, we first construct its subpaths as follows. For each node $h_j^{\lfloor\kappa\rfloor}$ in $\mathcal{H}^{\lfloor\kappa\rfloor}$ and any $0 < i \leq \Gamma$, we create a subpath of $\mathcal{P}^i$, denoted by $\mathcal{P}^i_j$, having ϕ'_j nodes. Denote nodes in $\mathcal{P}^i_j$ in order by $v^i_{j,1}, v^i_{j,2}, \ldots, v^i_{j,\phi'_j}$. We connect these paths together to form $\mathcal{P}^i_j$, i.e., for any $j \geq 0$, we create edges $(v^i_{j,\phi'_j}, v^i_{j+1,1})$ and $(v^i_{-j,\phi'_{-j}}, v^i_{-(j+1),1})$. Let

$$v^i_{-\infty} = v^i_{-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor},\phi'_{-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}} \quad \text{and} \quad v^i_{\infty} = v^i_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor},\phi'_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}}.$$

These two nodes can be thought of as the leftmost and rightmost nodes of path $\mathcal{P}^i$. We connect the paths together by adding edges between the leftmost (rightmost, respectively) nodes in the paths, i.e., for any i and i', we add edges $(v^i_{-\infty}, v^{i'}_{-\infty})$ ($(v^i_{\infty}, v^{i'}_{\infty})$, respectively).

We connect the highways and paths by adding an edge from each node $h_j^{\lfloor\kappa\rfloor}$ to $v^i_{j,1}$. We also create nodes s and t and connect s (t, respectively) to all nodes $v^i_{-\infty}$ (v^i_{∞}, respectively). See Fig. 8 for an example.

3.2.3 Description of $G_2(\Gamma, \Lambda, \kappa)$

We now modify $F(\Gamma, \kappa, \Lambda)$ to obtain $G_2(\Gamma, \Lambda, \kappa)$. Again, $G_2(\Gamma, \Lambda, \kappa)$ has three parameters, a real $\kappa \geq 1$ and two integers $\Gamma \geq 1$ and $\Lambda \geq 2$. The two basic units in the construction of $G_2(\Gamma, \Lambda, \kappa)$ are *highways* and *paths*. The highways are defined in

exactly the same way as before. The main modification is the definition of ϕ' (cf. Eq. (8)) which affects the number of nodes in the subpaths $\mathcal{P}^i_j$ of each path $\mathcal{P}^i$.

Definition of ϕ'. First, for a technical reason in the proof of Theorem 3.4, we need ϕ'_j to be small when $|j|$ is small. Thus, we define the following notation ϕ. For any j, define

$$\phi_j = \left\lfloor \frac{|j|}{\Lambda^{\lfloor \kappa \rfloor - 1}} \right\rfloor + 1 \,.$$

Note that ϕ_j can be viewed as the number of nodes in $\mathcal{H}_1$ with subscripts between 0 and j, i.e.,

$$\phi_j = \begin{cases} |\{h^1_{j'} \mid 0 \le j' \le j\}| & \text{if } j \ge 0 \\ |\{h^1_{j'} \mid j \le j' \le 0\}| & \text{if } j < 0 \,. \end{cases}$$

We now define ϕ' as follows. For any $j \ge 0$, let

$$\phi'_j = \phi'_{-j} = \min \left\{ \phi_j, \max(1, \lceil \lceil \kappa \rceil \Lambda^\kappa \rceil - \sum_{j' > j} \phi_{j'}) \right\} .$$

The reason we define ϕ' this way is that we use it to specify the number of nodes in the paths (as described in the previous section) and we want to be able to control this number precisely. In particular, while each path $\mathcal{P}^i$ in $F(\Gamma, \kappa, \Lambda)$ has $\Theta(\kappa \Lambda^{\lfloor \kappa \rfloor + 1})$ nodes (cf. Eq. (9)), the number of nodes in each path in $G_2(\Gamma, \Lambda, \kappa)$ is

$$\sum_{j=-\lceil \kappa \rceil \Lambda^{\lfloor \kappa \rfloor}}^{\lceil \kappa \rceil \Lambda^{\lfloor \kappa \rfloor}} \phi'_j = \Theta(\kappa \Lambda^\kappa). \tag{10}$$

We need this precision so that we can deal with any value of ℓ when we prove Theorem 4.1 in Section 4.2.

Finally, we make infinite copies of every edge except highway edges, i.e., those in $\cup_{i=1}^{\lfloor \kappa \rfloor} E(\mathcal{H}^i)$. (In other words, we make them have infinite capacity). As mentioned earlier, we do this so that we will have a freedom to specify the number of copies later on when we prove Theorem 4.1 in Section 4.2. Observe that if Theorem 3.4 holds then the same statement also holds when we set the number of edge copies of

each edge in $G_2(\Gamma, \Lambda, \kappa)$ to some specific numbers. Fig. 7(b) shows an example of $G_2(\Gamma, \Lambda, \kappa)$.

Now we are ready to prove Lemma 3.2.

Proof of Lemma 3.2. It follows from the construction of $G_2(\Gamma, \Lambda, \kappa)$ that the number of nodes in each path $\mathcal{P}^i$ is $\sum_{j=-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_j = \Theta(\kappa\Lambda^\kappa)$ (cf. Eq. (10)). Since there are Γ paths, the number of nodes in all paths is $\Theta(\Gamma\kappa\Lambda^\kappa)$. Each highway $\mathcal{H}^i$ has $2\lceil\kappa\rceil\Lambda^i + 1$ nodes. Therefore, there are $\sum_{i=1}^{\lfloor\kappa\rfloor}(2\lceil\kappa\rceil\Lambda^i + 1)$ nodes in the highways. For $\Lambda \geq 2$, the last quantity is $\Theta(\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor})$. Hence, the total number of nodes is $\Theta(\Gamma\kappa\Lambda^\kappa)$.

To analyze the diameter of $G_2(\Gamma, \Lambda, \kappa)$, observe that each node on any path $\mathcal{P}^i$ can reach a node in highway $\mathcal{H}^{\lfloor\kappa\rfloor}$ by traveling through $O(\kappa\Lambda)$ nodes in $\mathcal{P}^i$. Moreover, any node in highway $\mathcal{H}^i$ can reach a node in highway $\mathcal{H}^{i-1}$ by traveling trough $O(\Lambda)$ nodes in $\mathcal{H}^i$. Finally, there are $O(\kappa\Lambda)$ nodes in $\mathcal{H}^1$. Therefore, every node can reach any other node in $O(\kappa\Lambda)$ steps by traveling through $\mathcal{H}^1$. Note that this upper bound is tight since the distance between s and t is $\Omega(\kappa\Lambda)$. □

3.3 Proof of Theorem 3.3

3.3.1 Terminologies

For $1 \leq i \leq \lfloor(d^p - 1)/2\rfloor$, define the *i-left* and the *i-right* of the path $\mathcal{P}^\ell$ as

$$L_i(\mathcal{P}^\ell) = \{v_j^\ell \mid j \leq d^p - 1 - i\} \qquad \text{and} \qquad R_i(\mathcal{P}^\ell) = \{v_j^\ell \mid j \geq i\},$$

respectively. Define the i-left of the tree $\mathcal{T}$, denoted by $L_i(\mathcal{T})$, as the union of set $S = \{u_j^p \mid j \leq d^p - 1 - i\}$ and all ancestors of all vertices in S in $\mathcal{T}$. Similarly, the i-right $R_i(\mathcal{T})$ of the tree $\mathcal{T}$ is the union of set $S = \{u_j^p \mid j \geq i\}$ and all ancestors of all vertices in S. Now, the *i-left* and *i-right* sets of $G_1(\Gamma, p, d)$ are the union of those left and right sets,

$$L_i = \bigcup_\ell L_i(\mathcal{P}^\ell) \cup L_i(\mathcal{T}) \qquad \text{and} \qquad R_i = \bigcup_\ell R_i(\mathcal{P}^\ell) \cup R_i(\mathcal{T}).$$

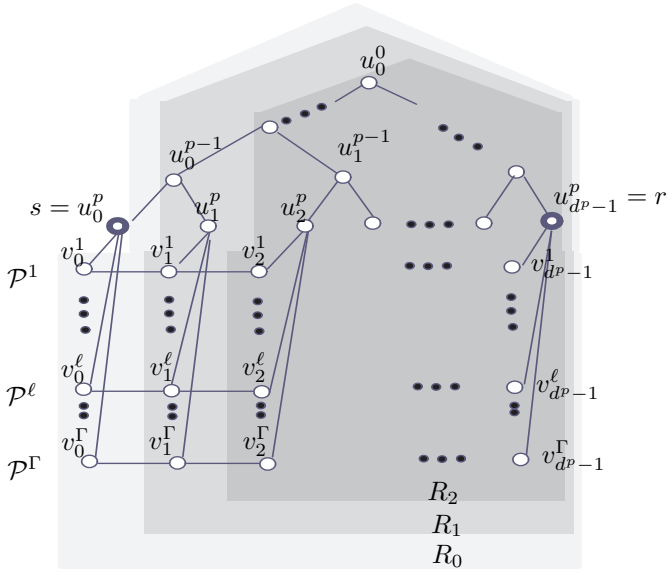


Figure 9: Examples of i-right sets.

For $i = 0$, the definition is slightly different; we set $L_0 = V \setminus \{r\}$ and $R_0 = V \setminus \{s\}$. See Figure 9.

Let $\mathcal{A}$ be any *deterministic* distributed algorithm run on network $G_1(\Gamma, p, d)$ for computing a function f. Fix any input strings x and y given to s and r respectively. Let $\varphi_{\mathcal{A}}(x, y)$ denote the execution of $\mathcal{A}$ on x and y. Denote the *state* of the vertex v at the end of round t during the execution $\varphi_{\mathcal{A}}(x, y)$ by $\sigma_{\mathcal{A}}(v, t, x, y)$. In two different executions $\varphi_{\mathcal{A}}(x, y)$ and $\varphi_{\mathcal{A}}(x', y')$, a vertex reaches the same state at time t (i.e., $\sigma_{\mathcal{A}}(v, t, x, y) = \sigma_{\mathcal{A}}(v, t, x', y')$) if and only if it receives the same sequence of messages on each of its incoming links.

For a given set of vertices $U = \{v_1, \ldots, v_\ell\} \subseteq V$, a *configuration*

$$C_{\mathcal{A}}(U, t, x, y) =< \sigma_{\mathcal{A}}(v_1, t, x, y), \ldots, \sigma_{\mathcal{A}}(v_\ell, t, x, y) >$$

is a vector of the states of the vertices of U at the end of round t of the execution $\varphi_{\mathcal{A}}(x, y)$. We note the following crucial observation used in [129] and many later results.

Observation 3.5. For any set $U \subseteq U' \subseteq V$, $C_{\mathcal{A}}(U, t, x, y)$ can be uniquely determined

by $C_{\mathcal{A}}(U', t-1, x, y)$ and all messages sent to U from $V \setminus U'$ at time t.

Proof. Recall that the state of each vertex v in U can be uniquely determined by its state $\sigma_{\mathcal{A}}(v, t-1, x, y)$ at time $t-1$ and the messages sent to it at time t. Moreover, the messages sent to v from vertices inside U' can be determined by $C_{\mathcal{A}}(U', t, x, y)$. Thus if the messages sent from vertices in $V \setminus U'$ are given then we can determine all messages sent to U at time t and thus we can determine $C_{\mathcal{A}}(U, t, x, y)$. □

From now on, to simplify notation, when $\mathcal{A}$, x and y are clear from the context, we use C_{L_t} and C_{R_t} to denote $C_{\mathcal{A}}(L_t, t, x, y)$ and $C_{\mathcal{A}}(R_t, t, x, y)$, respectively. The lemma below states that C_{L_t} (C_{R_t}, respectively) can be determined by $C_{L_{t-1}}$ ($C_{R_{t-1}}$, respectively) and dp messages generated by some vertices in R_{t-1} (L_{t-1} respectively) at time t. It essentially follows from Observation 3.5 and an observation that there are at most d^p edges linking between vertices in $V \setminus R_{t-1}$ ($V \setminus L_{t-1}$ respectively) and vertices in R_t (L_t respectively).

Lemma 3.6. *Fix any deterministic algorithm $\mathcal{A}$ and input strings x and y. For any $0 < t < (d^p - 1)/2$, there exist functions g_L and g_R, B-bit messages $M_1^{L_{t-1}}, \ldots, M_{dp}^{L_{t-1}}$ sent by some vertices in L_{t-1} at time t, and B-bit messages $M_1^{R_{t-1}}, \ldots, M_{dp}^{R_{t-1}}$ sent by some vertices in R_{t-1} at time t such that*

$$C_{L_t} = g_L(C_{L_{t-1}}, M_1^{R_{t-1}}, \ldots, M_{dp}^{R_{t-1}}), \text{ and} \tag{11}$$

$$C_{R_t} = g_R(C_{R_{t-1}}, M_1^{L_{t-1}}, \ldots, M_{dp}^{L_{t-1}}). \tag{12}$$

Proof. We prove Eq. (12) only. (Eq. (11) is proved in exactly the same way.) Observe that all neighbors of all path vertices in R_t are in R_{t-1}. Similarly, all neighbors of all leaf vertices in $V(\mathcal{T}) \cap R_t$ are in R_{t-1}. Moreover, for any non-leaf tree vertex u_i^ℓ (for some ℓ and i), if u_i^ℓ is in R_t then its parent and vertices $u_{i+1}^\ell, u_{i+2}^\ell, \ldots, u_{d^\ell-1}^\ell$ are in R_{t-1}. For any $\ell < p$ and t, let $u^\ell(R_t)$ denote the leftmost vertex that is at level ℓ of $\mathcal{T}$ and in R_t, i.e., $u^\ell(R_t) = u_i^\ell$ where i is such that $u_i^\ell \in R_t$ and $u_{i-1}^\ell \notin R_t$. (For

example, in Figure 9, $u^{p-1}(R_1) = u_0^{p-1}$ and $u^{p-1}(R_2) = u_1^{p-1}$.) Finally, observe that for any i and ℓ, if u_{i-1}^{ℓ} is in R_t then all children of u_i^{ℓ} are in R_t (otherwise, all children of u_{i-1}^{ℓ} are not in R_t and so is u_{i-1}^{ℓ}, a contradiction). Thus, all edges linking between vertices in R_t and $V \setminus R_{t-1}$ are in the following form: $(u^{\ell}(R_t), u')$ for some ℓ and child u' of $u^{\ell}(R_t)$.

Setting $U' = R_{t-1}$ and $U = R_t$ in Observation 3.5, we have that C_{R_t} can be uniquely determined by $C_{R_{t-1}}$ and messages sent to $u^{\ell}(R_t)$ from its children in $V \setminus R_{t-1}$. Note that each of these messages contains at most B bits since they correspond to a message sent on an edge in one round.

Observe further that, for any $t < (d^p - 1)/2$, $V \setminus R_{t-1} \subseteq L_{t-1}$ since L_{t-1} and R_{t-1} share some path vertices. Moreover, each $u^{\ell}(R_t)$ has d children. Therefore, if we let $M_1^{L_{t-1}}, \ldots, M_{dp}^{L_{t-1}}$ be the messages sent from children of $u^0(R_t), u^1(R_t), \ldots, u^{p-1}(R_t)$ in $V \setminus R_{t-1}$ to their parents (note that if there are less than dp such messages then we add some empty messages) then we can uniquely determine C_{R_t} by $C_{R_{t-1}}$ and $M_1^{L_{t-1}}, \ldots, M_{dp}^{L_{t-1}}$. Eq. (12) thus follows. □

Using the above lemma, we can now prove Theorem 3.3.

3.3.2 Proof

Proof of Theorem 3.3. Let f be the function in the theorem statement. Let $\mathcal{A}_\epsilon$ be any ϵ-error distributed algorithm for computing f on network $G_1(\Gamma, p, d)$. Fix a random string $\bar{r}$ used by $\mathcal{A}_\epsilon$ (shared by all vertices in $G_1(\Gamma, p, d)$) and consider the *deterministic* algorithm $\mathcal{A}$ run on the input of $\mathcal{A}_\epsilon$ and the fixed random string $\bar{r}$. Let $T_{\mathcal{A}}$ be the worst case running time of algorithm $\mathcal{A}$ (over all inputs). We only consider $T_{\mathcal{A}} < (d^p - 1)/2$, as assumed in the theorem statement. We show that Alice and Bob, when given $\bar{r}$ as the public random string, can simulate $\mathcal{A}$ using $2dpT_{\mathcal{A}}$ communication bits, as follows.

Alice and Bob make $T_{\mathcal{A}}$ *iterations* of communications. Initially, Alice computes

C_{L_0} which depends only on x. Bob also computes C_{R_0} which depends only on y. In each iteration $t > 0$, we assume that Alice and Bob know $C_{L_{t-1}}$ and $C_{R_{t-1}}$, respectively, before the iteration starts. Then, Alice and Bob will exchange at most $2dpB$ bits so that Alice and Bob know C_{L_t} and C_{R_t}, respectively, at the end of the iteration.

To do this, Alice sends to Bob the messages $M_1^{L_{t-1}}, \ldots, M_{dp}^{L_{t-1}}$ as in Lemma 3.6. Alice can generate these messages since she knows $C_{L_{t-1}}$ (by assumption). Then, Bob can compute C_{R_t} using Eq. (12) in Lemma 3.6. Similarly, Bob sends dp messages to Alice and Alice can compute C_{L_t}. They exchange at most $2dpB$ bits in total in each iteration since there are $2dp$ messages, each of B bits, exchanged.

After $T_{\mathcal{A}}$ iterations, Bob knows $C(R_{T_{\mathcal{A}}}, T_{\mathcal{A}}, x, y)$. In particular, he knows the output of $\mathcal{A}$ (output by r) since he knows the state of r after $\mathcal{A}$ terminates. He thus outputs the output of r.

Since $\mathcal{A}_\epsilon$ is ϵ-error, the probability (over all possible shared random strings) that $\mathcal{A}$ outputs the correct value of $f(x, y)$ is at least $1 - \epsilon$. Therefore, the communication protocol run by Alice and Bob is ϵ-error as well. Moreover, Alice and Bob communicates at most $2dpBT_{\mathcal{A}}$ bits. The theorem follows. □

3.4 Proof of Theorem 3.4

3.4.1 Terminologies

For any numbers i, j, i', and j', we say that $(i', j') \geq (i, j)$ if $i' > i$ or ($i' = i$ and $j' \geq j$). For any $-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor} \leq i \leq \lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}$ and $1 \leq j \leq \phi'_i$, define the (i, j)-*set* as

$$S_{i,j} = \begin{cases} \{h^x_{i'} \mid 1 \leq x \leq \kappa,\ i' \leq i\} \ \cup\ \{v^x_{i',j'} \mid 1 \leq x \leq \Gamma,\ (i,j) \geq (i',j')\} \cup \{s\} & \text{if } i \geq 0 \\ \{h^x_{i'} \mid 1 \leq x \leq \kappa,\ i' \geq i\} \ \cup\ \{v^x_{i',j'} \mid 1 \leq x \leq \Gamma,\ (-i,j) \geq (-i',j')\} \cup \{r\} & \text{if } i < 0\,. \end{cases}$$

See Figure 10 for an example. For convenience, for any $i > 0$, let $S_{i,0} = S_{i-1,\phi'_{i-1}}$ and $S_{-i,0} = S_{-(i-1),\phi'_{-(i-1)}}$, and, for any j, let $S_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}+1,j} = S_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor},\phi'_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}}$ and $S_{-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}-1,j} = S_{-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor},\phi'_{-\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}}}$.

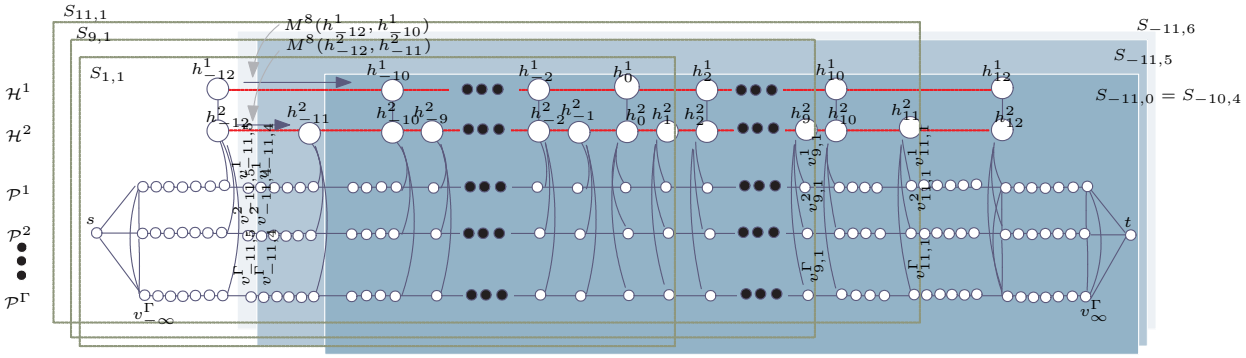


Figure 10: An example of round 11 in the proof of Theorem 3.4 (see detail in Example 3.9).

Let $\mathcal{A}$ be any *deterministic* distributed algorithm run on $G_2(\Gamma, \Lambda, \kappa)$ for computing a function f. Fix any input strings x and y given to s and t respectively. Let $\varphi_{\mathcal{A}}(x, y)$ denote the execution of $\mathcal{A}$ on x and y. Denote the *state* of the node v at the end of time τ during the execution $\varphi_{\mathcal{A}}(x, y)$ by $\sigma_{\mathcal{A}}(v, \tau, x, y)$. Let $\sigma_{\mathcal{A}}(v, 0, x, y)$ be the state of the node v before the execution $\varphi_{\mathcal{A}}(x, y)$ begins. Note that $\sigma_{\mathcal{A}}(v, 0, x, y)$ is independent of the input if $v \notin \{s, t\}$, depends only on x if $v = s$ and depends only on y if $v = t$. Moreover, in two different executions $\varphi_{\mathcal{A}}(x, y)$ and $\varphi_{\mathcal{A}}(x', y')$, a node reaches the same state at time τ (i.e., $\sigma_{\mathcal{A}}(v, \tau, x, y) = \sigma_{\mathcal{A}}(v, \tau, x', y')$) if and only if it receives the same sequence of messages on each of its incoming links.

For a given set of nodes $U = \{v_1, \ldots, v_\ell\} \subseteq V$, a *configuration*

$$C_{\mathcal{A}}(U, \tau, x, y) =< \sigma_{\mathcal{A}}(v_1, \tau, x, y), \ldots, \sigma_{\mathcal{A}}(v_\ell, \tau, x, y) >$$

is a vector of the states of the nodes of U at the end of time τ of the execution $\varphi_{\mathcal{A}}(x, y)$. From now on, to simplify notations, when $\mathcal{A}$, x and y are clear from the context, we use $C^\tau_{i,j}$ to denote $C_{\mathcal{A}}(S_{i,j}, \tau, x, y)$.

3.4.2 Proof

Let $G = G_2(\Gamma, \Lambda, \kappa)$. Let f be the function in the theorem statement. Let $\mathcal{A}_\epsilon$ be any ϵ-error distributed algorithm for computing f on G. Fix a random string $\bar{r}$ used

by $\mathcal{A}_\epsilon$ (shared by all nodes in G) and consider the *deterministic* algorithm $\mathcal{A}$ run on the input of $\mathcal{A}_\epsilon$ and the fixed random string $\bar{r}$. Let $T_\mathcal{A}$ be the worst case running time of algorithm $\mathcal{A}$ (over all inputs). We only consider $T_\mathcal{A} \leq \kappa\Lambda^\kappa$, as assumed in the theorem statement. We show that Alice and Bob, when given $\bar{r}$ as the public random string, can simulate $\mathcal{A}$ using $(2\kappa B)T_\mathcal{A}$ communication bits in $8T_\mathcal{A}/(\kappa\Lambda)$ rounds, as follows. (We provide an example in the end of this section.)

Rounds, Phases, and Iterations. For convenience, we will name the rounds backward, i.e., Alice and Bob start at round $\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}$ and proceed to round $\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}-1$, $\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}-2$, and so on. Each round is divided into two *phases*, i.e., when Alice sends messages and Bob sends messages (recall that Alice sends messages first in each iteration). Each phase of round r is divided into ϕ'_r *iterations*. Each iteration simulates one round of algorithm $\mathcal{A}$. We call the i^{th} iteration of round r when Alice (Bob, respectively) sends messages the *iteration* $I_{r,A,i}$ ($I_{r,B,i}$, respectively). Therefore, in each round r we have the following order of iterations: $I_{r,A,1}$, $I_{r,A,2}$, ..., I_{r,A,ϕ'_r}, $I_{r,B,1}$, ..., I_{r,B,ϕ'_r}. For convenience, we refer to the time before communication begins as round $\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}+1$ and let $I_{r,A,0} = I_{r+1,A,\phi'_{r+1}}$ and $I_{r,B,0} = I_{r+1,B,\phi'_{r+1}}$.

Our goal is to simulate one round of algorithm $\mathcal{A}$ per iteration. That is, after iteration $I_{r,B,i}$ finishes, we will finish the $(\sum_{r'=r+1}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_{r'} + i)^{th}$ round of algorithm $\mathcal{A}$. Specifically, we let

$$t_r = \sum_{r'=r+1}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_{r'}$$

and our goal is to construct a protocol with properties as in the following lemma.

Lemma 3.7. *There exists a protocol such that there are at most κB bits sent in each iteration and satisfies the following properties. For any $r \geq 0$ and $0 \leq i \leq \phi'_r$,*

1. *after $I_{r,A,i}$ finishes, Alice and Bob know $C^{t_r+i}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ and $C^{t_r+i}_{-r,\phi'_{-r}-i}$, respectively, and*

2. after $I_{r,B,i}$ finishes, Alice and Bob know $C^{t_r+i}_{r,\phi'_r-i}$ and $C^{t_r+i}_{-r+i\Lambda^{\lfloor\kappa\rfloor-1},1}$, respectively.

Proof. We first argue that the properties hold for iteration $I_{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}+1,A,0}$, i.e., before Alice and Bob starts communicating. After round $r = \lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}$ starts, Alice can compute $C^0_{r+1,0} = C^0_{r+1,1} = C^0_{r,\phi'_r}$ which contains the states of all nodes in $G_2(\Gamma, \Lambda, \kappa)$ except t. She can do this because every node except s and t has the same state regardless of the input and the state of s depends only on her input string x. Similarly, Bob can compute $C^0_{-(r+1),0} = C^0_{-(r+1),1} = C^0_{r,\phi'_r}$ which depends only on his input y.

Now we show that, if the lemma holds for any iteration $I_{r,A,i-1}$ then it also holds for iteration $I_{r,A,i}$ as well. Specifically, we show that if Alice and Bob know $C^{t_r+i-1}_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$ and $C^{t_r+i-1}_{-r,\phi'_{-r}-(i-1)}$, respectively, then they will know $C^{t_r+i}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ and $C^{t_r+i}_{-r,\phi'_{-r}-i}$, respectively, after Alice sends at most κB messages.

First we show that Alice can compute $C^{t_r+i}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ without receiving any message from Bob. Recall that Alice can compute $C^{t_r+i}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ if she knows

- $C^{t_r+i-1}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$, and
- all messages sent to all nodes in $S_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ at time $t_r + i$ of algorithm $\mathcal{A}$.

By assumption, Alice knows $C^{t_r+i-1}_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$ which implies that she knows $C^{t_r+i-1}_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ since $S_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1} \subseteq S_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$. Moreover, observe that all neighbors of all nodes in $S_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ are in $S_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$. Thus, Alice can compute all messages sent to all nodes in $S_{r-i\Lambda^{\lfloor\kappa\rfloor-1},1}$ at time $t_r + i$ of algorithm $\mathcal{A}$. Therefore, Alice can compute $C^{t_r+i}_{r+i\Lambda^{\lfloor\kappa\rfloor-1},1}$ without receiving any message from Bob.

Now we show that Bob can compute $C^{t_r+i}_{-r,\phi'_{-r}-i}$ by receiving at most κB bits from Alice and use the knowledge of $C^{t_r+i-1}_{-r,\phi'_{-r}-i+1}$. Note that Bob can compute $C^{t_r+i}_{-r,\phi'_{-r}-i}$ if he knows

- $C^{t_r+i-1}_{-r,\phi'_{-r}-i}$, and
- all messages sent to all nodes in $S_{-r,\phi'_{-r}-i}$ at time $t_r + i$ of algorithm $\mathcal{A}$.

By assumption, Bob knows $C^{t_r+i-1}_{-r,\phi'_{-r}-i+1}$ which implies that he knows $C^{t_r+i-1}_{-r,\phi'_{-r}-i}$ since $S_{-r,\phi'_{-r}-i} \subseteq S_{-r,\phi'_{-r}-i+1}$. Moreover, observe that all neighbors of all nodes in $S_{-r,\phi'_{-r}-i}$ are in $S_{-r,\phi'_{-r}-i+1}$, except

$$h^{\lfloor\kappa\rfloor}_{-(r+1)},\ h^{\lfloor\kappa\rfloor-1}_{-(\lfloor r/\Lambda\rfloor+1)},\ \ldots,\ h^{\lfloor\kappa\rfloor-i}_{-(\lfloor r/\Lambda^i\rfloor+1)},\ \ldots,\ h^{1}_{-(\lfloor r/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor+1)}.$$

In other words, Bob can compute all messages sent to all nodes in $S_{-r,\phi'_{-r}-i}$ at time $t_r + i$ except

$$M^{t_r+i}(h^{\lfloor\kappa\rfloor}_{-(r+1)}, h^{\lfloor\kappa\rfloor}_{-r}),\ \ldots,\ M^{t_r+i}(h^{\lfloor\kappa\rfloor-i}_{-(\lfloor r/\Lambda^i\rfloor+1)}, h^{\lfloor\kappa\rfloor-i}_{-\lfloor r/\Lambda^i\rfloor}),\ \ldots,$$
$$M^{t_r+i}(h^{1}_{-(\lfloor r/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor+1)}, h^{1}_{-(\lfloor r/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor)})$$

where $M^{t_r+i}(u,v)$ is the message sent from u to v at time t_r+i of algorithm $\mathcal{A}$. Observe further that Alice can compute these messages because she knows $C^{t_r+i-1}_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$ which contains the states of $h^{\lfloor\kappa\rfloor}_{-(r+1)},\ \ldots,\ h^{\lfloor\kappa\rfloor-i}_{-(\lfloor r/\Lambda^i\rfloor+1)},\ \ldots,\ h^{1}_{-(\lfloor r/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor+1)}$ at time $t_r + i - 1$. (In particular, $C^{t_r+i-1}_{r-(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$ is a superset of $C^{t_r+i-1}_{0,1}$ which contains the states of $h^{\lfloor\kappa\rfloor}_{-(r+1)},\ \ldots,\ h^{1}_{-(\lfloor r/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor+1)}$.) So, Alice can send these messages to Bob and Bob can compute $C^{t_r+i}_{-r,\phi'_{-r}-i}$ at the end of the iteration. Each of these messages contains at most B bits since each of them corresponds to a message sent on one edge. Therefore, Alice sends at most κB bits to Bob in total. This shows the first property.

After Alice finishes sending messages, the two parties will switch their roles and a similar protocol can be used to show that the second property, i.e., if the lemma holds for any iteration $I_{r,B,i-1}$ then it also holds for iteration $I_{r,B,i}$ as well. That is, if Alice and Bob know $C^{t_r+i-1}_{r,\phi'_r-(i-1)}$ and $C^{t_r+i-1}_{-r+(i-1)\Lambda^{\lfloor\kappa\rfloor-1},1}$, respectively, then Bob can send κB bits to Alice so that they can compute $C^{t_r+i}_{r,\phi'_r-i}$ and $C^{t_r+i}_{-r+i\Lambda^{\lfloor\kappa\rfloor-1},1}$, respectively. □

Let P be the protocol as in Lemma 3.7. Alice and Bob will run protocol P until round r', where r' is the largest number such that $t_{r'} + \phi'_{r'} \geq T_{\mathcal{A}}$. Lemma 3.7 implies that after iteration $I_{r',B,T_{\mathcal{A}}-t_{r'}}$, Bob knows $C^{t_{-r'}+T_{\mathcal{A}}-t_{r'}}_{-r',\phi'_{-r'}-T_{\mathcal{A}}+t_{r'}} = C^{T_{\mathcal{A}}}_{-r',\phi'_{-r'}-T_{\mathcal{A}}+t_{r'}}$ (note

that $\phi'_{-r'} - T_{\mathcal{A}} + t_{r'} \geq 0$). In particular, Bob knows the state of node t at time $T_{\mathcal{A}}$, i.e., he knows $\sigma_{\mathcal{A}}(t, T_{\mathcal{A}}, x, y)$. Thus, Bob can output the output of $\mathcal{A}$ which is output from t.

Since $\mathcal{A}_\epsilon$ is ϵ-error, the probability (over all possible shared random strings) that $\mathcal{A}$ outputs the correct value of $f(x, y)$ is at least $1 - \epsilon$. Therefore, the communication protocol run by Alice and Bob is ϵ-error as well. The number of rounds is bounded as in the following claim.

Claim 3.8. *If algorithm $\mathcal{A}$ finishes in time $T_{\mathcal{A}} \leq \lceil\kappa\rceil\Lambda^\kappa$ then $r' > \lceil\kappa\rceil\Lambda^\kappa - 8T_{\mathcal{A}}/(\lceil\kappa\rceil\Lambda)$. In other words, the number of rounds Alice and Bob need to simulate $\mathcal{A}$ is $8T_{\mathcal{A}}/(\lceil\kappa\rceil\Lambda)$*

Proof. Let $R^* = 8T_{\mathcal{A}}/(\lceil\kappa\rceil\Lambda)$ and let $r^* = \Lambda^{\lfloor\kappa\rfloor} - R^* + 1$. Assume for the sake of contradiction that Alice and Bob need more than R^* rounds. This means that $r' < r^*$. Alice and Bob requiring more than R^* rounds implies that

$$\sum_{r=r^*}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_r = t_{r^*} + \phi'_{r^*} < T_{\mathcal{A}} \leq \lceil\kappa\rceil\Lambda^\kappa \,. \tag{13}$$

It follows that for any $r \geq r^*$,

$$\begin{aligned}
\phi'_r &= \min\left(\phi(h_r^{k'}), \max(1, \lceil\lceil\kappa\rceil\Lambda^\kappa\rceil - \sum_{r'>r}\phi_{r'})\right) && \text{(by definition of } \phi'_r) \\
&= \phi_r && \text{(because } \textstyle\sum_{r\geq r^*}\phi'_r < \lceil\kappa\rceil\Lambda^\kappa) \\
&= \left\lfloor \frac{r}{\Lambda^{\lfloor\kappa\rfloor-1}} \right\rfloor + 1 && \text{(by definition of } \phi_r).
\end{aligned}$$

Therefore, the total number of steps that can be simulated by Alice and Bob up to

round r^* is

$$\begin{aligned}
\sum_{r=r^*}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \phi'_r &= \sum_{r=r^*}^{\lceil\kappa\rceil\Lambda^{\lfloor\kappa\rfloor}} \left(\left\lfloor \frac{r}{\Lambda^{\lfloor\kappa\rfloor-1}} \right\rfloor + 1\right) \\
&\geq \Lambda^{\lfloor\kappa\rfloor-1} \sum_{i=1}^{\lfloor R^*/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor} (\lceil\kappa\rceil\Lambda - i) \\
&\geq \Lambda^{\lfloor\kappa\rfloor-1} \cdot \frac{\lfloor R^*/\Lambda^{\lfloor\kappa\rfloor-1}\rfloor(\lceil\kappa\rceil\Lambda - 1)}{2} \\
&\geq \frac{R^*\lceil\kappa\rceil\Lambda}{8} \\
&\geq T_{\mathcal{A}}
\end{aligned}$$

contradicting Eq. (13). □

Since there are at most κB bits sent in each iteration and Alice and Bob runs P for $T_{\mathcal{A}}$ iterations, the total number of bits exchanged is at most $(2\kappa B)T_{\mathcal{A}}$. This completes the proof of Theorem 3.4.

Example 3.9. Figure 10 shows an example of the protocol we use above. Before iteration $I_{11,A,1}$ begins, Alice and Bob know $C^7_{11,1}$ and $C^7_{-11,5}$, respectively (since Alice and Bob already simulated $\mathcal{A}$ for $\phi'_{12} = 7$ steps in round 12). Then, Alice computes and sends $M^8(h^2_{-12}, h^2_{-11})$ and $M^8(h^1_{-12}, h^1_{-10})$ to Bob. Alice and Bob then compute $C^8_{11,1}$ and $C^8_{-11,6}$, respectively, at the end of iteration $I_{11,A,1}$. After they repeat this process for five more times, i.e. Alice sends

$$M^9(h^2_{-12}, h^2_{-11}), M^{10}(h^2_{-12}, h^2_{-11}), \ldots, M^{13}(h^2_{-12}, h^2_{-11}), \quad \text{and}$$

$$M^9(h^1_{-12}, h^1_{-10}), M^{10}(h^1_{-12}, h^1_{-10}), \ldots, M^{13}(h^1_{-12}, h^1_{-10}),$$

Bob will be able to compute $C^{13}_{-11,0} = C^{13}_{-10,4}$. Note that Alice is able to compute $C^8_{9,1}$, $C^9_{7,1}$, …, $C^{12}_{1,1}$ without receiving any messages from Bob so she can compute and send the previously mentioned messages to Bob.

Related publications. The preliminary versions of this chapter appeared as part of the joint results with Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Gopal Pandurangan, David Peleg, and Roger Wattenhofer [41, 120].

CHAPTER IV

A TIGHT LOWER BOUND ON DISTRIBUTED RANDOM WALK COMPUTATION

In this chapter, we show a tight lower bound for computing a random walk. Recall the following definitions from Chapter 1. In the 1-RW-DoS (one random walk where destination outputs source) version, we are given a network $G = (V, E)$ and a source node $s \in V$. The goal is to devise a distributed algorithm such that, in the end, some node v outputs the ID of s, where v is a destination node picked according to the probability that it is the destination of a random walk of length ℓ starting at s. For the 1-RW-SoD (source outputs destination) version, we want s to output the ID of v instead. Finally, for the 1-RW-pos (nodes know their positions) version, we want each node to know its position(s) in the random walk.

Also recall that, in Chapter 2, we present algorithms that solve 1-RW-DoS, 1-RW-SoD, and 1-RW-pos in $\tilde{O}(\sqrt{\ell D})$ time (cf. Lemma 2.6). In this chapter, we show an *unconditional* lower bound of $\Omega(\sqrt{\ell D} + D)$ for all three versions of the random walk computation problem. This means that the algorithms in Chapter 2 are optimal for all three variations. In particular, we show the following theorem.

Theorem 4.1. *For any n, D, B and ℓ such that $D \leq \ell \leq (n/(D^3 B))^{1/4}$, there exists a family of n-node networks of diameter D in the B-model such that performing a random walk (for any of the three versions) of length $\Theta(\ell)$ on these networks requires $\Omega(\sqrt{\ell D} + D)$ rounds.*

To prove this theorem, we use the connection between *bounded-round* communication complexity and distributed algorithm lower bounds shown in Chapter 3. A particular communication complexity problem that we will use is the following *pointer*

chasing problem.

4.1 The pointer chasing problem

In this section, we define the pointer chasing problem and prove its lower bound (Lemma 4.3) which will be used to prove Theorem 4.1 in the next section.

Informally, the r-round pointer chasing problem has parameters r and m and there are two players, which could be Alice and Bob or nodes s and t, who receive functions $f_A : [m] \to [m]$ and $f_B : [m] \to [m]$, respectively. The goal is to compute a function starting from 1 and alternatively applying f_A and f_B for r times each, i.e., compute $f_B(\ldots f_A(f_B(f_A)))$ where f_A and f_B appear r times each. To be precise, let $\mathcal{F}_m$ be the set of functions $f : [m] \to [m]$. For any $i \geq 0$ define $g^i : \mathcal{F}_m \times \mathcal{F}_m \to [m]$ inductively as

$$g^0(f_A, f_B) = 1 \quad \text{and} \quad g^i(f_A, f_B) \begin{cases} f_A(g^{i-1}(f_A, f_B)) & \text{if } i > 0 \text{ and } i \text{ is odd,} \\ f_B(g^{i-1}(f_A, f_B)) & \text{if } i > 0 \text{ and } i \text{ is even.} \end{cases}$$

Also define function $\textsc{pc}^{i,m}(f_A, f_B) = g^{2i}(f_A, f_B)$. The goal of the r-round pointer chasing problem is to compute $\textsc{pc}^{r,m}(f_A, f_B)$.

Observe that if Alice and Bob can communicate for r rounds then they can compute $\textsc{pc}^{r,m}$ naively by exchanging $O(r \log m)$ bits. Interestingly, Nisan and Wigderson [121] show that if Alice and Bob are allowed only $r-1$ rounds then they essentially cannot do anything better than having Alice sent everything she knows to Bob.[1]

Theorem 4.2. *[121]* $R_{1/3}^{(r-1)-cc-pub}(\textsc{pc}^{r,m}) = \Omega(m/r^2 - r \log m)$.

The pointer chasing problem on $G_2(\Gamma, \Lambda, \kappa)$. We now consider the pointer chasing problem on network $G_2(\Gamma, \Lambda, \kappa)$ where s and t receive f_A and f_B respectively. The following lemma follows from Theorem 3.4 and 4.2.

[1] In fact this holds even when Alice and Bob are allowed r rounds but Alice cannot send a message in the first round.

Lemma 4.3. *For any κ, Γ, $\Lambda \geq 2$, $m \geq \kappa^2\Lambda^{4\kappa}B$, $16\Lambda^{\kappa-1} \geq r > 8\Lambda^{\kappa-1}$, $R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) = \Omega(\kappa\Lambda^{\kappa})$.*

Proof. If $R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) > \kappa\Lambda^{\kappa}$ then we are done so we assume that $R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) \leq \kappa\Lambda^{\kappa}$. Thus,

$$R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) \geq R_{1/3}^{\frac{8R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m})}{\kappa\Lambda}-cc-pub}(\text{PC}^{r,m})/(2\kappa B) \tag{14}$$

$$\geq R_{1/3}^{\frac{8\kappa\Lambda^{\kappa}}{\kappa\Lambda}-cc-pub}(\text{PC}^{r,m})/(2\kappa B) \tag{15}$$

$$= \Omega((m(8\Lambda^{\kappa-1})^{-2} - 8\Lambda^{\kappa-1}B)/(\kappa B)) \tag{16}$$

$$= \Omega(\kappa\Lambda^{\kappa}) \tag{17}$$

where Eq. (14) is by Theorem 3.4 and the fact that $R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) \leq \kappa\Lambda^{\kappa}$, Eq. (15) uses the fact that the communication does not increase when we allow more rounds and $R_{1/3}^{G_2(\Gamma,\Lambda,\kappa),s,t}(\text{PC}^{r,m}) \leq \kappa\Lambda^{\kappa}$, Eq. (16) follows from Theorem 4.2 with the fact that $16\Lambda^{\kappa-1} \geq r > 8\Lambda^{\kappa-1}$ and Eq. (17) is because $m \geq \kappa^2\Lambda^{4\kappa}B$. □

4.2 Proof of Theorem 4.1

We will prove the theorem only for the version where destination outputs source (1-RW-DoS). This is because we can convert algorithms for the other two versions to solve this version by adding $O(D)$ rounds. To see this, observe that once the source outputs the ID of the destination, we can take additional $O(D)$ rounds to send the ID of the source to the destination. Similarly, if nodes know their positions, the node with position ℓ can output the source's ID by taking additional $O(D)$ rounds to request for the source's ID.

Now, for the lower bound of 1-DoS, we first show the $\Omega(D)$ lower bound which is fairly straightforward.

Lemma 4.4. *For every $D \leq \ell \leq n$, there exists a graph G of diameter D such that any distributed algorithm that solves 1-RW-DoS on G uses $\Omega(D)$ rounds with high probability.*

Proof. Let s and t be two nodes of distance exactly D from each other and with only this one path between them. A walk of length ℓ starting at s has a non-zero probability of ending up at t. In this case, for the source ID of s to reach t, at least D rounds of communication will be required. Using multi-edges, one can force, with high probability, the traversal of the random walk to be along this path of length D. □

The rest of this chapter focuses on showing the $\Omega(\sqrt{\ell D})$ lower bound. Theorem 4.1, for the case where destination outputs source, follows from the following lemma.

Lemma 4.5. *For any real $\kappa \geq 1$ and integers $\Lambda \geq 2$, and $\Gamma \geq 32\kappa^2\Lambda^{6\kappa-1}\log n$, there exists a family of networks $\mathcal{H}$ such that any network $H \in \mathcal{H}$ has $\Theta(\kappa\Gamma\Lambda^\kappa)$ nodes and diameter $D = \Theta(\kappa\Lambda)$, and any algorithm for computing the destination of a random walk of length $\ell = \Theta(\Lambda^{2\kappa-1})$ requires $\Omega(\sqrt{\ell D})$ time on some network $H \in \mathcal{H}$.*

Proof. We show how to compute $\textsc{pc}^{r,m}$ on $G = G_2(\Gamma, \Lambda, \kappa)$ by reducing the problem to the problem of sampling a random walk destination in some network H_{f_A,f_B}, obtained by restrict the number of copies of some edges in G, depending on input functions f_A and f_B. We let $\mathcal{H}$ be the family of network H_{f_A,f_B} over all input functions. Note that for any input functions, an algorithm on H_{f_A,f_B} can be run on G with the same running time since every edge in G has more capacity than its counterpart in H_{f_A,f_B}.

Let $r = 16\Lambda^{\kappa-1}$ and $m = \kappa^2\Lambda^{5\kappa}\log n$. Note that $2rm \leq \Gamma$. For any $i \leq r$ and $j \leq m$, let

$$S^{i,j} = \mathcal{P}^{2(i-1)m+j} \quad \text{and} \quad T^{i,j} = \mathcal{P}^{2(i-1)m+m+j}\,.$$

That is, $S^{1,1} = \mathcal{P}^1$, …, $S^{1,m} = \mathcal{P}^m$, $T^{1,1} = \mathcal{P}^{m+1}$, …, $T^{1,m} = \mathcal{P}^{2m}$, $S^{2,1} = \mathcal{P}^{2m+1}$, …, $T^{r,m} = \mathcal{P}^{2rm}$. Let L be the number of nodes in each path. Note that $L = \Theta(\kappa\Lambda^\kappa)$ by Lemma 3.2. Denote the nodes in $S^{i,j}$ from *left to right* by $s_1^{i,j}, \ldots, s_L^{i,j}$. (Thus, $s_1^{i,j} = v_{-\infty}^{2(i-1)m+j}$ and $s_L^{i,j} = v_{\infty}^{2(i-1)m+j}$.) Also denote the nodes in $T^{i,j}$ from *right to*

left by $t_1^{i,j}, \ldots, t_L^{i,j}$. (Thus, $t_1^{i,j} = v_\infty^{2(i-1)m+m+j}$ and $t_L^{i,j} = v_{-\infty}^{2(i-1)m+m+j}$.) Note that for any i and j, $s_1^{i,j}$ and $t_L^{i,j}$ are adjacent to s while $s_L^{i,j}$ and $t_1^{i,j}$ are adjacent to t.

Now we construct H_{f_A,f_B}. For simplicity, we fix input functions f_A and f_B and denote H_{f_A,f_B} simply by H. To get H we let every edge in G have one copy (thus with capacity $O(\log n)$), except the following edges. For any $i \le r$, $j \le m$, and $x < L$, we have $(6\Gamma\ell)^{2(i-1)L+x}$ copies of edges between nodes $s_x^{i,j}$ and $s_{x+1}^{i,j}$ and $(6\Gamma\ell)^{2(i-1)L+L+x}$ copies of edges between nodes $t_x^{i,j}$ and $t_{x+1}^{i,j}$. Note that these numbers of copies of edges are always the same, regardless of the input f_A and f_B.

Additionally, we have the following numbers of edges which depend on the input functions. First, s specifies the following number of edges between its neighbors. For any $i \le r$, $j \le m$, we have $(6\Gamma\ell)^{2(i-1)L+L}$ copies of edges between nodes $t_L^{i,j}$ and $s_1^{i,f_A(j)}$. These numbers of edges can be specified in one round since both $s_1^{i,j}$ and $t_L^{i,f_A(j)}$ are adjacent to s. Similarly, we have $(6\Gamma\ell)^{2(i-1)L+2L}$ copies of edges between nodes $t_1^{i,j}$ and $s_L^{i+1,f_B(j)}$ which can be done in one round since both nodes are adjacent to t. This completes the description of H.

Now we use any random walk algorithm to compute the destination of a walk of length $\ell = 2rL - 1 = \Theta(\Lambda^{2\kappa-1})$ on H by starting a random walk at $s_1^{1,f(A)}$. If the random walk destination is $t_L^{r,j}$ for some j, then node t outputs the number j; otherwise, node t outputs an arbitrary number.

Claim 4.6. *Node t outputs $\textsc{pc}^{r,m}(f_A, f_B)$ with probability at least $2/3$.*

Proof. Let P^* be the path consisting of nodes $s_1^{1,f_A(1)}$, …, $s_L^{1,f_A(1)}$, $t_1^{1,f_B(f_A(1))}$, …, $t_L^{1,f_B(f_A(1))}$, $s_1^{1,f_A(f_B(f_A(1)))}$, …, $s_L^{i,g^{2i-1}(f_A,f_B)}$, $t_1^{i,g^{2i}(f_A,f_B)}$, …, $t_L^{r,g^{2r}(f_A,f_B)}$. We claim that the random walk will follow path P^* with probability at least $2/3$. The node of distance $(2rL-1)$ from $s_1^{1,f_A(1)}$ in this path is $t_L^{r,g^{2r}(f_A,f_B)} = t_L^{r,\mathrm{PC}^{r,m}(1)}$ and thus the algorithm described above will output $\textsc{pc}^{r,m}(1)$ with probability at least $2/3$.

To prove the above claim, consider any node u in path P^*. Let u' and u'' be

the node before and after u in P^*, respectively. Let m' and m'' be the number of multiedges (u, u') and (u, u''), respectively. Observe that $m'' \geq 6\Gamma\ell m'$. Moreover, observe that there are at most Γ edges between u and other nodes. Thus, if a random walk is at u, it will continue to u'' with probability at least $1 - \frac{1}{3\ell}$. By union bound, the probability that a random walk will follow P^* is at least $1 - \frac{1}{3}$, as claimed. □

Thus, if there is any random walk algorithm with running time $O(T)$ on all networks in $\mathcal{H}$ then we can use such algorithm to solve $\text{PC}^{r,m}$ (with error probability $1/3$) in time $O(T)$. Using the lower bound of computing solving $\text{PC}^{r,m}$ in Lemma 4.3, the random walk computation also has a lower bound of $\Omega(\kappa\Lambda^\kappa) = \Omega(\sqrt{\ell D})$ as claimed. □

To prove Theorem 4.1 with the given parameters n, D and ℓ, we simply set Λ and κ so that $\kappa\Lambda = D$ and $\Lambda^{2\kappa-1} = \Theta(\ell)$. This choice of Λ and κ exists since $\ell \geq D$. Setting Γ large enough so that $\Gamma \geq 32\kappa^2\Lambda^{6\kappa-1}\log n$ while $\Gamma = \Theta(n)$. (This choice of Γ exists since $\ell \leq (n/(D^3 \log n))^{1/4}$.) By applying the above lemma, Theorem 4.1 follows.

4.3 Conclusions

In this chapter we prove a tight unconditional lower bound on the time complexity of distributed random walk computation, implying that the algorithms in Chapter 2 for generating a random walk are time optimal. To the best of our knowledge, this is the first lower bound that the diameter plays a role of multiplicative factor. Our proof technique comes from associating the bounded-round communication complexity to the distributed algorithm lower bounds, with network diameter as a trade-off factor. The weaker form (without the presence of rounds) are also shown to have many applications, as we will show in the next chapter.

There are still some problems left open. One interesting open problem is showing a lower bound of performing a long walk and many walks. For example, one can

generate a *random spanning tree* by computing a walk of length equals the cover time (using the version where every node knows their positions, 1-RW-pos). It is interesting to see if performing such a walk can be done faster. Additionally, the upper and lower bounds of the problem of generating a random spanning tree itself is very interesting since its current upper bound of $\tilde{O}(\sqrt{m}D)$ (cf. Chapter 2) simply follows as an application of random walk computation while no lower bound is known.

Related publications. The preliminary version of this chapter appeared as a joint work with Atish Das Sarma and Gopal Pandurangan [120].

CHAPTER V

DISTRIBUTED VERIFICATION AND HARDNESS OF DISTRIBUTED APPROXIMATION

In this chapter, we study many "global" graph problems in distributed networks. We show that, for many fundamental graph problems, verification is as hard as optimization. An important consequence of this is that many optimization problems, such as minimum spanning tree, cannot be done faster even when we allow approximation.

5.1 Overview of Technical Approach

We prove our lower bounds by establishing an interesting connection between communication complexity and distributed computing, as shown in Chapter 3. Our lower bound proofs consider the network $G_1(\Gamma, p, d)$ as defined in Chapter 3. This network is a result of a developments through a series of papers in the literature [55, 107, 129]. However, while previous results [55, 129] rely on counting the number of states to analyze the *mailing problem* (along with some sophisticated techniques for the variant, called *corrupted mail problem*, in the case of approximation algorithm lower bounds) and use Yao's method [151] (with appropriate input distributions) to get lower bounds for randomized algorithms, our results are achieved using the following three steps of simple reductions.

First, as we showed in Chapter 3, we reduce the lower bounds of problems in the standard communication complexity model [95] to the lower bounds of the equivalent problems in the "distributed version" of communication complexity. Specifically, we relate the *communication* lower bound from the standard communication complexity model [95] to compute some appropriately chosen function f, to the distributed

time complexity lower bound for computing the same function in a specially chosen graph G. In the standard model, Alice and Bob can communicate directly (via a bidirectional edge of bandwidth one). In the distributed model, we assume that Alice and Bob are some vertices of G and they together wish to compute the function f using the communication graph G. The choice of graph G is critical. We use a graph called $G_1(\Gamma, p, d)$ (parameterized by Γ, d and p) that was first used in [55]. We show a reduction from the standard model to the distributed model, the proof of which relies on certain observations similar to those used in previous results (e.g., [129]).

Second, the connection established in the first step allows us to bypass the state counting argument and Yao's method, and reduces our task in proving lower bounds of verification problems to merely picking the right function f to reduce from. The function f that is useful in showing our randomized lower bounds is the *set disjointness function*, which is the quintessential problem in the world of communication complexity with applications to diverse areas and has been studied for decades (see a recent survey in [29]). Following the result well known in communication complexity [95], we show that the distributed version of this problem has an $\Omega(\sqrt{n/(B \log n)})$ lower bound on graphs of small diameter. We then reduce this problem to the verification problems using simple reductions similar to those used in data streams [76]. The set disjointness function yields randomized lower bounds and works for many problems (see Figure 11), but it does not reduce to certain other problems such as spanning tree. To show lower bounds for this problem, we use a different function f called *Hamiltonian cycle*. However, this reduction yields only a *one-sided error* randomized lower bound for the corresponding verification problems.

Finally, we reduce the verification problem to hardness of distributed approximation for a variety of problems to show that the same lower bounds hold for approximation algorithms as well. For this, we use a reduction whose idea is similar to one used to prove hardness of approximating TSP (Traveling Salesman Problem) on general

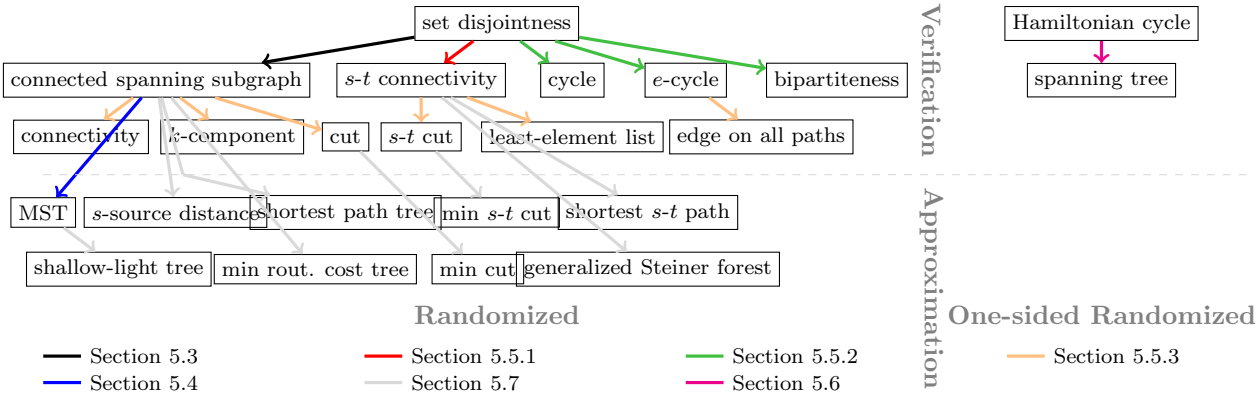


Figure 11: Problems and reductions between them to obtain randomized and one-sided error randomized lower bounds. For all problems, we obtain lower bounds as in Figure 1.

graphs (see, e.g., [144]): We convert a verification problem to an optimization problem by introducing edge weight in such a way that there is a large gap between the optimal values for the cases where H satisfies, or does not satisfy a certain property. This technique is surprisingly simple, yet yields strong unconditional hardness bounds — many hitherto unknown, left open (e.g., minimum cut) [52] and some that improve over known ones (e.g., MST and shortest path tree) [55]. As mentioned earlier, our approach shows that approximating MST by *any* factor needs $\tilde{\Omega}(\sqrt{n})$ time, while the previous result due to Elkin gave a bound that depends on α (the approximation factor), i.e. $\tilde{\Omega}(\sqrt{n/\alpha})$, using more sophisticated techniques.

Figure 11 summarizes these reductions that will be proved in this chapter.

To highlight the main ideas, we first show the proof of the lower bound of approximating the minimum spanning tree in Sections 5.2, 5.3 and 5.4. In the later sections, we show the rest lower bounds.

5.2 The Set-Disjointness problem

To prove the theorem, we need the lower bound for computing *set disjointness function*.

Definition 5.1 (Set Disjointness function)**.** Given two b-bit strings x and y, the *set*

disjointness function, denoted by $\texttt{disj}(x, y)$, is defined to be 1 if the inner product $< x, y >$ is 0 (i.e., $x_i = 0$ or $y_i = 0$ for every $1 \leq i \leq b$) and 0 otherwise. We refer to the problem of computing `disj` function on $G_1(\Gamma, p, d)$ on Γ-bit input strings given to s and r by $\texttt{DISJ}(G_1(\Gamma, p, d), s, r, \Gamma)$.

The following lemma is a consequence of Theorem 3.3 and the communication complexity lower bound of computing `disj`.

Lemma 5.2. *For any Γ, d, p, there exists a constant $\epsilon > 0$ such that any ϵ-error algorithm solving $\texttt{DISJ}(G_1(\Gamma, p, d), s, r, \Gamma)$ requires $\Omega(\min(d^p, \frac{\Gamma}{dpB}))$ time.*

Proof. If $R_\epsilon^{G_1(\Gamma,p,d),s,r}(\texttt{disj}) \geq (d^p - 1)/2$ then $R_\epsilon^{G_1(\Gamma,p,d),s,r}(\texttt{disj}) = \Omega(d^p)$ and we are done. Otherwise, Theorem 3.3 implies that $R_\epsilon^{cc-pub}(\texttt{disj}) \leq 2dpB \cdot R_\epsilon^{G_1(\Gamma,p,d),s,r}(\texttt{disj})$. Now we use the fact that $R_\epsilon^{cc-pub}(\texttt{disj}) = \Omega(\Gamma)$ for the function `disj` on Γ-bit inputs, for some $\epsilon > 0$ [13, 81, 16, 131] (also see [95, Example 3.22] and references therein). It follows that $R_\epsilon^{G_1(\Gamma,p,d),s,r}(\texttt{disj}) = \Omega(\Gamma/(dpB))$. □

5.3 Randomized Lower Bounds for Distributed Verification

In this section, we present randomized lower bounds for many verification problems for graph of various diameters, as shown in Figure 1. The general theorem is below. The highlight the main ideas, in this section we prove the theorem only for the spanning connected subgraph verification problem. This will be useful later in proving many hardness of approximation results, including approximating the minimum spanning tree. In this problem, we want to verify whether H is connected and spans all nodes of G, i.e., every node in G is incident to some edge in H. Definitions of other problems and proofs of their lower bounds are in Section 5.5.

Theorem 5.3. *For any $p \geq 1$, $B \geq 1$, and $n \in \{2^{2p+1}pB, 3^{2p+1}pB, \ldots\}$, there exists a constant $\epsilon > 0$ such that any ϵ-error distributed algorithm for any of the following problems requires $\Omega((n/(pB))^{\frac{1}{2} - \frac{1}{2(2p+1)}})$ time on some $\Theta(n)$-vertex graph of diameter*

$2p + 2$ *in the B model: Spanning connected subgraph, connectivity, s-t connectivity, k-components, bipartiteness, cycle containment, e-cycle containment, cut, s-t cut, least-element list [34, 86], and edge on all paths.*

In particular, for graphs with diameter $D = 4$, we get $\Omega((n/B)^{1/3})$ lower bound and for graphs with diameter $D = \log n$ we get $\Omega(\sqrt{n/(B \log n)})$. Similar analysis also leads to a $\Omega(\sqrt{n/B})$ lower bound for graphs of diameter n^δ for any $\delta > 0$, and $\Omega((n/B)^{1/4})$ lower bound for graphs of diameter 3 using the same analysis as in [55]. We note that the lower bound holds even in the public coin model where every vertex shares a random string.

The lower bound of spanning connected subgraph verification essentially follows from the following lemma.

Lemma 5.4. *For any Γ, $d \geq 2$ and p, there exists a constant $\epsilon > 0$ such that any ϵ-error distributed algorithm for spanning connected subgraph verification on graph $G_1(\Gamma, p, d)$ can be used to solve the $\texttt{DISJ}(G_1(\Gamma, p, d), s, t, \Gamma)$ problem on $G_1(\Gamma, p, d)$ with the same time complexity.*

Proof. Consider an ϵ-error algorithm $\mathcal{A}$ for the spanning connected subgraph verification problem, and suppose that we are given an instance of the $\texttt{DISJ}(G_1(\Gamma, p, d), s, t, \Gamma)$ problem with input strings x and y. We use $\mathcal{A}$ to solve this instance of set disjointness problem as follows.

First, we mark all path edges and tree edges as participating in H. All spoke edges are marked as not participating in subgraph H, except those incident to s and r for which we do the following: For each bit x_i, $1 \leq i \leq \Gamma$, vertex s indicates that the spoke edge (s, v_0^i) participates in H if and only if $x_i = 0$. Similarly, for each bit y_i, $1 \leq i \leq \Gamma$, vertex r indicates that the spoke edge $(r, v_{d^p-1}^i)$ participates in H if and only if $y_i = 0$. (See Figure 12.)

Note that the participation of all edges, except those incident to s and r, is decided independently of the input. Moreover, one round is sufficient for s and r to inform

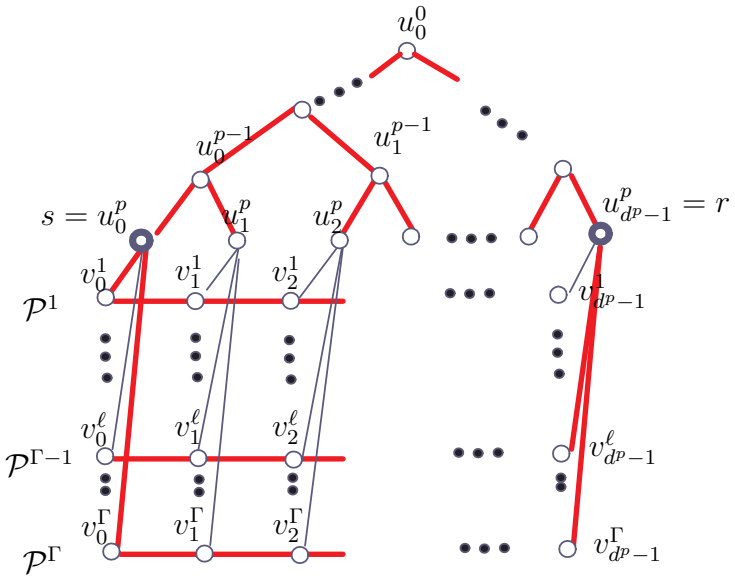


Figure 12: Example of H for the spanning connected subgraph problem (marked with thick red edges) when $x = 0...10$ and $y = 1...00$.

their neighbors the participation of edges incident to them. Hence, one round is enough to construct H. Then, algorithm $\mathcal{A}$ is started.

Once algorithm $\mathcal{A}$ terminates, vertex r determines its output for the set disjointness problem by stating that both input strings are disjoint if and only if spanning connected subgraph verification algorithm verified that the given subgraph H is indeed a spanning connected subgraph.

Observe that H is a spanning connected subgraph if and only if for all $1 \leq i \leq \Gamma$ at least one of the edges (s, v_0^i) and $(r, v_{d^p-1}^i)$ is in H; thus, by the construction of H, H is a spanning connected subgraph if and only if the input strings x, y are disjoint, i.e., for every i either $x_i = 0$ or $y_i = 0$. Hence the resulting algorithm has correctly solved the given instance of the set disjointness problem. □

Using Lemma 5.2, we obtain the following result.

Corollary 5.5. *For any Γ, d, p, there exists a constant $\epsilon > 0$ such that any ϵ-error algorithm for spanning connected subgraph verification problem requires $\Omega(\min(d^p, \frac{\Gamma}{dpB}))$ time on some $\Theta(\Gamma d^p)$-vertex graph of diameter $2p + 2$.*

In particular, if we consider $\Gamma = d^{p+1}pB$ then $\Omega(\min(d^p, \Gamma/(dpB))) = \Omega(d^p)$. Moreover, by Lemma 3.1, $G(d^{p+1}pB, d, p)$ has $n = \Theta(d^{2p+1}pB)$ vertices and thus the lower bound $\Omega(d^p)$ becomes $\Omega((n/(pB))^{\frac{1}{2}-\frac{1}{2(2p+1)}})$. Theorem 5.3 (for the case of spanning connected subgraph) follows.

5.4 Hardness of Distributed Approximation

In this section we show a time lower bound of $\Omega(\sqrt{n/(B\log n)})$ for approximation algorithms of many problems. For distributed approximation problems such as MST, we assume that a weight function $\omega : E \to \mathbb{R}^+$ associated with the graph assigns a nonnegative real weight $\omega(e)$ to each edge $e = (u, v) \in E$. Initially, the weight $\omega(e)$ is known only to the adjacent vertices, u and v. We assume that the edge weights are bounded by a polynomial in n (the number of vertices). It is assumed that B is large enough to allow the transmission of any edge weight in a single message.

We show the hardness of distributed approximation for many problems, as in the theorem below. To highlight the main ideas, we only prove the theorem for the minimum spanning tree problem here. Definitions and proofs of other problems can be found in Section 5.7.

Theorem 5.6. *For any polynomial function $\alpha(n)$, numbers p, $B \geq 1$, and $n \in \{2^{2p+1}pB, 3^{2p+1}pB, \ldots\}$, there exists a constant $\epsilon > 0$ such that any $\alpha(n)$-approximation ϵ-error distributed algorithm for any of the following problems requires $\Omega((\frac{n}{pB})^{\frac{1}{2}-\frac{1}{2(2p+1)}})$ time on some $\Theta(n)$-vertex graph of diameter $2p+2$ in the B model: minimum spanning tree [55, 129], shortest s-t path, s-source distance [53], s-source shortest path tree [55], minimum cut [52], minimum s-t cut, maximum cut, minimum routing cost spanning tree [150], shallow-light tree [128], and generalized Steiner forest [86].*

Recall that in the minimum spanning tree problem, we are given a connected graph G and we want to compute the minimum spanning tree (i.e., the spanning tree of minimum weight). At the end of the process each vertex knows which edges

incident to it are in the output tree.

Recall the following standard notions of an *approximation algorithm*. We say that a randomized algorithm $\mathcal{A}$ is *α-approximation ϵ-error* if, for any input instance $\mathcal{I}$, algorithm $\mathcal{A}$ outputs a solution that is at most α times the optimal solution of $\mathcal{I}$ with probability at least $1 - \epsilon$. Therefore, in the minimum spanning tree, an α-approximation ϵ-error algorithm should output a number that is at most α times the total weight of the minimum spanning tree, with probability at least $1 - \epsilon$.

Proof of Theorem 5.6 for the case of minimum spanning tree. Let $\mathcal{A}_\epsilon$ be an $\alpha(n)$-approximation ϵ-error algorithm for the minimum spanning tree problem. We show that $\mathcal{A}_\epsilon$ can be used to solve the spanning connected subgraph verification problem using the same running time.

To do this, construct a weight function on edges in G, denoted by ω, by assigning weight 1 to all edges in H and $n\alpha(n)$ to all other edges. Note that constructing ω does not need any communication since each vertex knows which edges incident to it are in H. Now we find the weight W of the minimum spanning tree using $\mathcal{A}_\epsilon$ and announce that H is a spanning connected subgraph if and only if W is less than $n\alpha(n)$.

Now we show that the weighted graph (G, ω) has a spanning tree of weight less than $n\alpha(n)$ if and only if H is a spanning connected subgraph of G and thus the algorithm above is correct: Suppose that H is a spanning connected subgraph. Then, there is a spanning tree that is a subgraph of H and has weight $n - 1 < n\alpha(n)$. Thus the minimum spanning tree has weight less than $n\alpha(n)$. Conversely, suppose that H is not a spanning connected subgraph. Then, any spanning tree must contain an edge not in H. Therefore, any spanning tree has weight at least $n\alpha(n)$ as claimed. □

5.5 The rest randomized lower bounds

In this section, we show the randomized lower bounds as claimed in Theorem 5.3 for the following problems (listed in Figure 11).

Definition 5.7 (Problems with randomized lower bounds)**.** We define:

- s-t **connectivity verification problem**: In addition to G and H, we are given two vertices s and t (s and t are known by every vertex). We would like to verify whether s and t are in the same connected component of H. (Section 5.5.1.)
- **cycle containment verification problem:** We want to verify if H contains a cycle. (Section 5.5.2.)
- e**-cycle containment verification problem:** Given an edge e in H (known to vertices adjacent to it), we want to verify if H contains a cycle containing e. (Section 5.5.2.)
- **bipartiteness verification problem**: We want to verify whether H is bipartite. (Section 5.5.2.)
- **connectivity verification problem**: We want to verify whether H is connected. We also consider the k**-component verification problem** where we want to verify whether H has at most k connected components. (Note that k is not part of the input so 2-component and 3-component problems are different problems.) The connectivity verification problem is the special case where $k = 1$. (Section 5.5.3)
- **cut verification problem**: We want to verify whether H is a cut of G, i.e., G is not connected when we remove edges in H. (Section 5.5.3)
- s-t **cut verification problem**: We want to verify whether H is an s-t cut, i.e., when we remove all edges E_H of H from G, we want to know whether s and t

are in the same connected component or not. (Section 5.5.3)

- **least-element list verification problem [34, 86]:** Given a distinct rank (integer) $r(v)$ to each node v in the weighted graph G, for any nodes u and v, we say that v is the *least element* of u if v has the lowest rank among vertices of distance at most $d(u, v)$ from u. Here, $d(u, v)$ denotes the weighted distance between u and v. The *Least-Element List* (LE-list) of a node u is the set $\{< v, d(u, v) > \mid v \text{ is the least element of } u \}$. (Section 5.5.3)

  In the least-element list verification problem, each vertex knows its rank as an input, and some vertex u is given a set $S = \{< v_1, d(u, v_1) >, < v_2, d(u, v_2) > , \ldots\}$ as an input. We want to verify whether S is the least-element list of u. (Section 5.5.3)

- **edge on all paths verification problem:** Given nodes u, v and edge e. We want to verify whether e lies on all paths between u, v in H. (Section 5.5.3)

Lower bounds of the above problems are stated in Theorem 5.3 and the reductions are summarized in Figure 11.

5.5.1 Randomized lower bound of s-t connectivity verification

Similar to the lower bound of the spanning connected subgraph verification problem, the lower bounds of s-t connectivity follow from the following lemma.

Lemma 5.8. *For any Γ, $d \geq 2$ and p, there exists a constant $\epsilon > 0$ such that any ϵ-error distributed algorithm for s-t connectivity verification problem on graph $G_1(\Gamma, p, d)$ can be used to solve the $\mathtt{DISJ}(G_1(\Gamma, p, d), s, r, \Gamma)$ problem on $G_1(\Gamma, p, d)$ with the same time complexity.*

Proof. We use the same argument as in the proof of Lemma 5.4 except that we construct the subgraph H as follows.

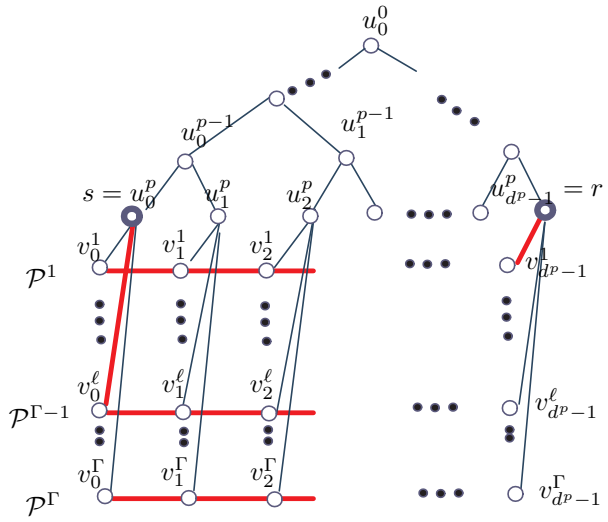


Figure 13: Example of H for s-t connectivity problem (marked with thick red edges) when $x = 0...10$ and $y = 1...00$.

s-t verification: First, all path edges are marked as participating in subgraph H. All tree edges are marked as not participating in H. All spoke edges, except those incident to s and r, are also marked as not participating. For each bit x_i, $1 \leq i \leq \Gamma$, vertex s indicates that the spoke edge (s, v_0^i) participates in H if and only if $x_i = 1$. Similarly, for each bit y_i, $1 \leq i \leq \Gamma$, vertex r indicates that the spoke edge $(r, v_{d^p-1}^i)$ participates in H if and only if $y_i = 1$. (See Figure 13.)

Once algorithm $\mathcal{A}_{st}$ terminates, vertex r determines its output for the set disjointness problem by stating that both input strings are disjoint if and only if s-r connectivity verification algorithm verified that s and r are *not* connected in the given subgraph.

For the correctness of this algorithm, observe that s and r are connected in H if and only if there exists $1 \leq i \leq \Gamma$ such that both edges $(v_0^i, s), (v_{d^p-1}^i, r)$ are in H; thus, by the construction of the s-r connected subgraph candidate H, H is s-r connected if and only if the input strings x and y are *not* disjoint, i.e., there exists i such that $x_i = 1$ and $y_i = 1$. Hence the resulting algorithm has correctly solved the given instance of the set disjointness problem. □

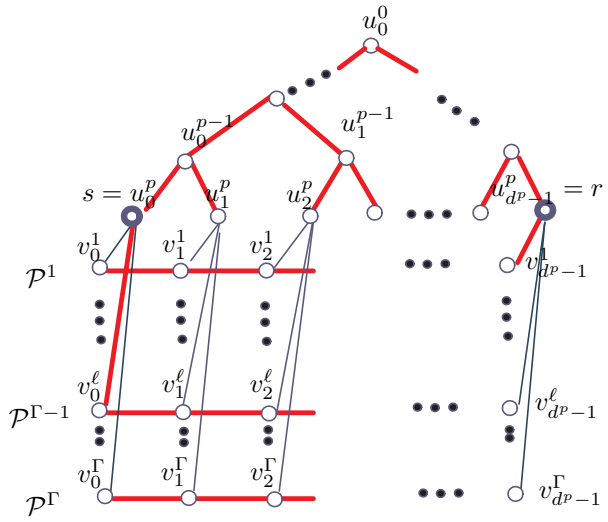


Figure 14: Example of H for the cycle and e-cycle containment and bipartiteness verification problem when $x = 0...10$ and $y = 1...00$.

5.5.2 A randomized lower bound for cycle containment, e-cycle containment, and bipartiteness verification problem

Lemma 5.9. *There exists a constant $\epsilon > 0$ such that any ϵ-error distributed algorithm for cycle containment, e-cycle containment, or bipartiteness verification problem on graph $G_1(\Gamma, p, d)$ can be used to solve the $\mathtt{DISJ}(G_1(\Gamma, p, d), s, r, \Gamma)$ problem on $G_1(\Gamma, p, d)$ with the same time complexity.*

Proof. cycle verification problem: We construct H in the same way as in the proof of Lemma 5.8 *except* that the tree edges are participating in H (see Figure 14).

In the case that the input strings are disjoint, H will consist of the tree connecting s and r as well as 1) paths connected to s but not to r and 2) paths connected to r but not to s and 3) paths connected neither to r nor s. Thus there is no cycle in H. In the case that the input strings are not disjoint, we let i be an index that makes them not disjoint, that is $x_i = y_i = 1$. This causes a cycle in H consisting of some tree edges and path P^i that are connected by edges (s, v_0^i) and $(v_{d^p-1}^i, r)$ at their endpoints. Thus we have the following claim.

Claim 5.10. *H contains a cycle if and only if the input strings are not disjoint.*

e-cycle containment verification problem: We use the previous construction for H and let e be the tree edge adjacent to s (i.e., e connects s to its parent). Observe that, in this construction, H contains a cycle if and only if H contains a cycle containing e. Therefore, we have the following claim.

Claim 5.11. *e is contained in a cycle in H if and only if the input strings are not disjoint.*

bipartiteness verification problem: Finally, we can verify if such an edge e is contained in a cycle by verifying the bipartiteness. First, we replace $e = (s, u_0^{p-1})$ by a path (s, v', u_0^{p-1}), where v' is an additional/virtual vertex. This can be done without changing the input graph G by having vertex s simulated algorithms on both s and v'. The communication between s and v' can be done internally. The communication between v' and u_0^{p-1} can be done by s. We construct H' the same way as H with both (s, v') and (v', u_0^{p-1}) marked as participating. The lower bound of bipartite follows from this claim.

Like in the previous proofs, we observe that if the input strings are not disjoint, then either H or H' are not bipartite. We consider two cases: when d^p is even and odd. When d^p is even and the input strings are not disjoint, there exists i such that there is a cycle in H consisting of some tree edges (including e) and path P_i that are connected by edges (s, v_0^i) and $(v_{d^p-1}^i, r)$ at their endpoints. This cycle is of length $2p + (d^p - 1) + 2$ – an odd number causing H to be not bipartite. If d^p is odd, then by the same argument there is an odd cycle of length $(2p + 1) + (d^p - 1) + 2$ in H' (this cycle includes the edges (s, v') and (v', u_0^{p-1}) that replaces e); thus H' is not bipartite.

Now we consider the converse: If the input strings are disjoint, then H does not contain a cycle by the argument of the proof of the cycle containment problem (which uses the same graph). In follows that H' does not contain a cycle as well. Therefore, we have the following claim.

Claim 5.12. *H and H' are both bipartite if and only if the input strings are disjoint.*

□

5.5.3 Randomized lower bounds of connectivity, k-component, cut, s-t cut, least-element list, and edge on all paths verification

connectivity verification problem: We reduce from the spanning connected subgraph verification problem. Let $\mathcal{A}(G, H)$ be an algorithm that verifies if H is connected in $O(\tau(n))$ time on any n-vertex graph G and subgraph H, we show that there is an algorithm $\mathcal{A}'(G', H')$ that verifies whether H' is a spanning connected subgraph in $O(\tau(n') + D')$ time, where n' and D' is the number of vertices in G' and its diameter, respectively. Thus, the lower bounds (which are larger than D) of the spanning connected subgraph problem apply to the connectivity verification problem as well.

To do this, recall that, by definition, H' is a spanning connected subgraph if and only if every node is incident to at least one edge in H' and H' is connected. Verifying that every node is incident to at least one edge in H' can be done in $O(D)$ rounds and checking if H' is connected can be done in $O(\tau(n'))$ rounds by calling $\mathcal{A}(G, H)$ with $H = H'$ and $G = G'$. The total running time of $\mathcal{A}'$ is thus $O(\tau(n') + D)$.

k-component verification problem: The above argument can be extended to show the lower bound of k-component problem, as follows. Suppose again that we want to check if H is a spanning connected subgraph. Now we add $k - 1$ virtual nodes adjacent to some node s in G. These nodes are added to H (denote the resulting subgraph by H') but will not be incident to any edges in H' and are simulated by s. Observe that the new graph, say G', has diameter $D' = D + 1$ and the number of nodes is $n' = n + k \leq 2n$. Moreover, H is a spanning tree of G if and only if H' has k connected component in G' (the spanning subgraph H of G plus $k - 1$ single nodes). Therefore, if we can check if H has at most k (k constant) connected component in G' in $O(\tau(n'))$ time then we can also check if H is a spanning connected subgraph in G.

cut verification problem: We again reduce from the spanning connected subgraph problem. Recall the definition of the cut that H is a cut if and only if G is *not* connected when we remove edges in H. In other words, H is a cut if and only if $\bar{H}$ is not a spanning connected subgraph of G where $\bar{H}$ is the graph resulting from removing edges in H.

Thus, given a subgraph H', we verify if H' is a spanning connected subgraph as follows. Let H'' be the graph obtained by removing edges $E(H')$ of H' from G. Recall again that H' is a spanning connected subgraph if and only if H'' is not a cut. Thus, we verify if H'' is a cut. We announce that H' is a spanning connected subgraph if and only if H'' is verified not to be a cut.

s-t cut verification problem: The lower bound of s-t cut is proved similarly: H' is s-t connected if and only if H'' obtained by removing edges in H' from G is *not* an s-t cut.

Least-element list verification problem: We reduce from s-t connectivity. We set the rank of s to 0 and the rank of other nodes to any distinct positive integers. Assign weight 0 to all edges in H and 1 to other edges. Give a set $S = \{< s, 0 >\}$ to vertex t. Then we verify if S is the least-element list of t. Observe that if s and t are connected by H then the distance between them must be 0 and thus S is the least-element list of t. On the other hand, if s and t are not connected then the distance between them will be at least 1 and S will not be the least-element list of t.

Edge on all paths verification problem: We reduce from the e-cycle containment problem using the following observation: H does not contain a cycle containing e if and only if e lies on all paths between u and v in H where $e = uv$.

5.6 One-sided Error Lower Bound of Spanning Tree Verification

We show the following lower bound of one-sided error randomized algorithms for the spanning tree verification problem. Let us first define the notion of one-sided error.

Definition 5.13 (One-sided error algorithms)**.** We say that an algorithm $\mathcal{A}$ computes a function f with *one-sided ϵ-error* if for every input x such that $f(x) = 0$, $\mathcal{A}$ outputs 0 with probability one, and for every input x such that $f(x) = 1$, $\mathcal{A}$ outputs 1 with probability at least $1 - \epsilon$. We let $R^1_\epsilon(f)$ denote the one-sided ϵ-error of computing f.

Recall that, in the ST verification problem, we would like to verify whether H is a spanning tree of G. Our result is the following.

Theorem 5.14. *For any $0 \leq \epsilon < 1$, p, $B \geq 1$, and $n \in \{2^{2p+1}pB, 3^{2p+1}pB, \ldots\}$, any distributed algorithm for ST verification problem requires $\Omega((\frac{n \log\log n}{pB})^{\frac{1}{2} - \frac{1}{2(2p+1)}})$ time on some $\Theta(n)$-vertex graph of diameter $O(2p+2)$ in the B model.*

In particular, this result shows that any one-sided error algorithms for ST verification requires $\tilde{\Omega}(\sqrt{n})$ time. This is the first lower bound of randomized algorithms for both ST and MST verification. (Prior to this, only lower bounds deterministic algorithms for MST verification are known [90].)

We do this by reducing from the communication complexity of the *Hamiltonian cycle* problem.

5.6.1 A one-sided randomized lower bound of Hamiltonian cycle problem

Definition 5.15 (Hamiltonian cycle)**.** There are two b-element sets of vertices, denoted by $u_1, \ldots, u_b$ and $v_1, \ldots, v_b$. Alice and Bob each gets a perfect matching between vertices in these two sets. That is, Alice and Bob get

$$X = \{u_{i_1}v_{i_1}, u_{i_2}v_{i_2}, \ldots, u_{i_b}v_{i_b}\} \quad \text{and} \quad Y = \{u_{j_1}v_{j_1}, u_{j_2}v_{j_2}, \ldots, u_{j_b}v_{j_b}\},$$

respectively. They want to check whether $X \cup Y$ forms a Hamiltonian cycle or not. We let ham be a function such that $\mathsf{ham}(X, Y)$ is one if $X \cup Y$ forms a Hamiltonian cycle and zero otherwise. We denote this problem by HAM.

The lower bound of HAM problem essentially follows from the result of Raz and Spieker [130], as follows.

Lemma 5.16. *For any $\epsilon > 0$, any one-sided ϵ-error protocol requires $\Omega(b \log\log b)$ bits of communication. That is, $R^1_\epsilon = \Omega(b \log\log b)$.*

Proof. We use the result of Raz and Spieker [130] which states that *non-deterministic* communication complexity of Ham is $\Omega(b \log\log b)$. In notation, $N^1(\mathsf{ham}) = \Omega(b \log\log b)$. (Informally, $N^1(\mathsf{ham})$ denote the size of proof given to Alice and Bob and bits exchange between them in order to compute ham. We refer to [95] for the full definition of N^1.) Using the fact that $R^1_\epsilon(f) \geq N^1_\epsilon(f)$ for any function f and $0 \leq \epsilon < 1$ (see, e.g., [95, Proposition 3.7], the lemma follows. □

Lemma 5.17. *For any $0 \leq \epsilon < 1$, Γ, d, p, and $b = \Theta(\Gamma)$, any one-sided ϵ-error algorithm solving $\textsc{Ham}(G_1(\Gamma, p, d), s, r, \Gamma)$ requires $\Omega(\min(d^p, \Gamma \log\log \Gamma / dpB))$ time.*

Proof. We use the fact that $R_0^{cc-pub}(\mathsf{ham}) = \Omega(\Gamma \log\log \Gamma)$ for the function ham on Γ-bit inputs which follows from Lemma 5.16. Thus by Theorem 3.3, $R_\epsilon^{G_1(\Gamma,p,d),s,r}(f) = \Omega(\min d^p, \Gamma \log\log \Gamma / dpB)$ implying the lemma. □

5.6.2 Proof of Theorem 5.14 (sketched)

First, we modify the graph slightly by adding edges between leftmost (rightmost, respectively) nodes of all paths. Theorem 3.3 can still be proved in exactly the same way. Thus, Lemma 5.17 still holds on this modified graph. From now on, we abuse the notation and call this graph $G_1(\Gamma, p, d)$.

Consider a one-sided ϵ-error algorithm $\mathcal{A}$ for the ST verification problem, and suppose that we are given an instance of the $\textsc{Ham}(G_1(\Gamma, p, d), s, t, \Gamma)$ problem with input sets X and Y. We use $\mathcal{A}$ to solve this instance of the HAM problem as follows.

First, we mark all path edges and tree edges as participating in H. All spoke edges are marked as not participating in subgraph H, except those incident to s and r. It is left to mark edges incident to s and r and those between end vertices of paths.

Let $\Gamma = 2b - 1$. Define the following function g_A which maps nodes in the graph given to Alice and Bob to $s, v_0^1, v_0^2, \ldots, v_0^\Gamma$, as follows. We let $g_A(u_1), \ldots, g_A(u_b)$ be $s, v_0^1, v_0^2, \ldots, v_0^{b-1}$, respectively. We also let $g_A(v_1), \ldots, g_A(v_b)$ be $v_0^b, v_0^{b+1}, \ldots, v_0^\Gamma$, respectively. Now, if there is an edge between u_i and v_j in G_A (the graph given to Alice) then we mark an edge between $g_A(u_i)$ and $g_A(v_j)$ as participating; otherwise, we mark such edge as not participating. (For example, there is an edge between u_1 and v_b in G_A in Figure 15. Thus, we mark an edge between s and v_0^Γ as participating.)

Similarly, define the following function g_B which maps nodes in the graph given to Alice and Bob to $t, v_{d^p-1}^1, v_{d^p-1}^2, \ldots, v_{d^p-1}^\Gamma$, as follows. We let $g_B(u_1), \ldots, g_B(u_b)$ be $t, v_{d^p-1}^1, v_{d^p-1}^2, \ldots, v_{d^p-1}^{b-1}$, respectively. We also let $g_B(v_1), \ldots, g_B(v_b)$ be $v_{d^p-1}^b, v_{d^p-1}^{b+1}, \ldots, v_{d^p-1}^\Gamma$, respectively. Now, if there is an edge between u_i and v_j in G_B (the graph given to Bob) then we mark an edge between $g_B(u_i)$ and $g_B(v_j)$ as participating; otherwise, we mark such edge as not participating. There is one exception: If such edge is incident to r then we mark it as not participating. (For example, there is an edge between u_1 and v_1 in G_B in Figure 15. However, we mark an edge between r and $v_{d^p-1}^1$ as not participating as an exception.)

Note that the participation of all edges, except those incident to s and r and the end vertices of all paths, is decided independently of the input. Moreover, two rounds are sufficient for s and r to mark all these edges. Then, we start algorithm $\mathcal{A}$.

Once algorithm $\mathcal{A}$ terminates, vertex r determines its output for the HAM problem by stating that $X \cup Y$ form a Hamiltonian cycle if and only if spanning tree verification algorithm verified that the given subgraph H is indeed a spanning tree.

Observe that H is a spanning tree if and only if $X \cup Y$ is form a connected component, i.e., a Hamiltonian cycle. Hence the resulting algorithm has correctly

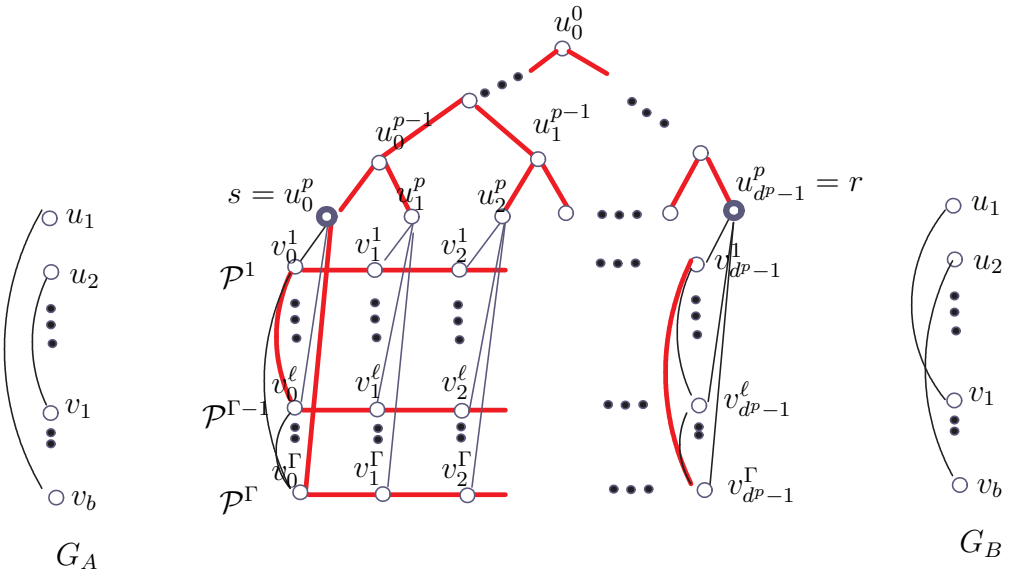


Figure 15: Example of H for the ST verification problem (marked with thick red edges) when the input graph G_A and G_B is as shown.

solved the given instance of the set disjointness problem.

Remark. We note that one can also show one-sided error randomized lower bounds of the following problem by reducing from communication complexity of the Hamiltonian cycle problem.

- **Hamiltonian cycle:** Given a graph G and subgraph H of G, we would like to verify whether H is a Hamiltonian cycle of G, i.e., H is a simple cycle of length n.

- **Simple path verification** Given a graph G, subgraph H of G verify that H is a simple path.

The main trick is that, instead of marking all tree edges as participating, we add some edges to the tree in such a way that they form a path from s to r that covers all tree nodes (see Fig. 16). This modification can be made while Theorem 3.3 is still maintained. The rest of the reduction is the same (see Fig. 17).

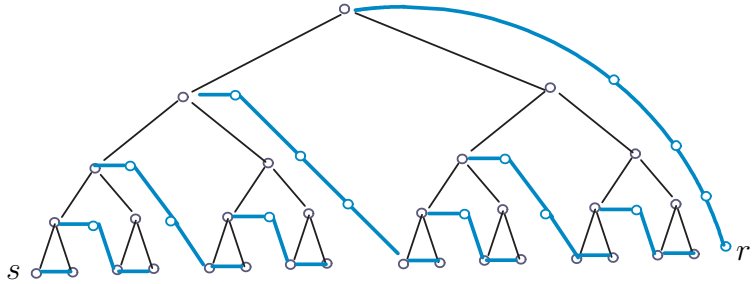


Figure 16: Example of the modification of the tree-part of G. Edges and nodes in blue are added to the tree.

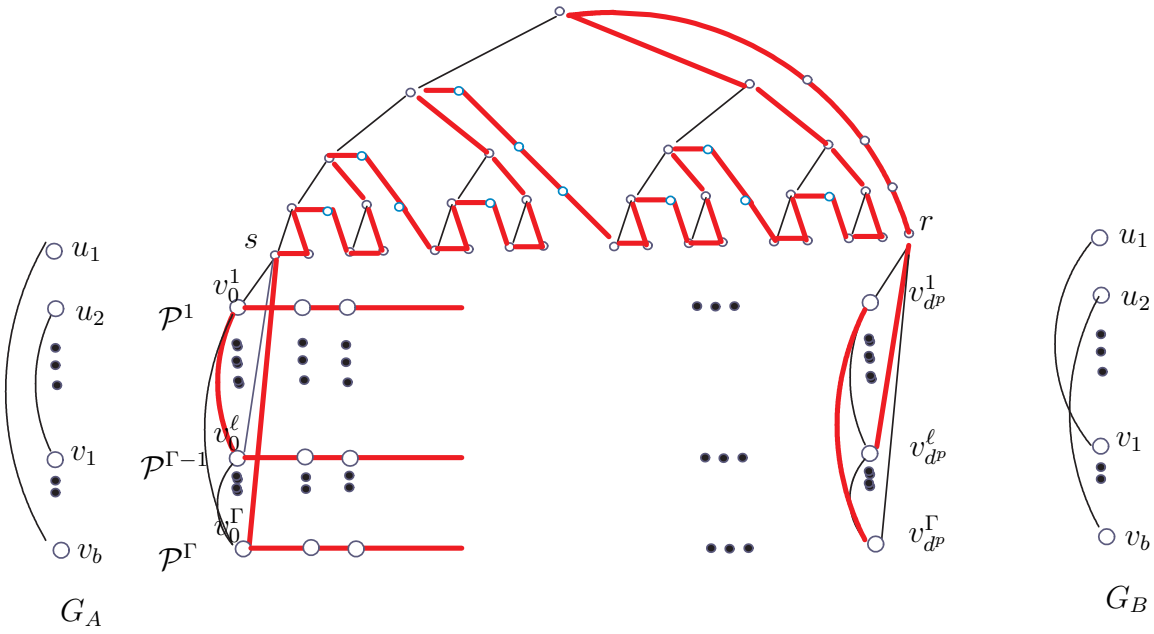


Figure 17: Example of the reduction from the communication complexity of the Hamiltonian cycle problem to the distributed Hamiltonian cycle verification problem. Red edges form subgraph H.

5.7 Details of hardness of approximation

In this section, we show the randomized lower bounds as claimed in Theorem 5.6 for the following problems (listed in Figure 11).

Problems: Given a connected graph G with a weight function ω on edges (where, for each edge e, $\omega(e)$ is known to nodes incident to e), we consider the following problems.

- In the **minimum spanning tree** problem [55, 129], we want to compute the weight of the minimum spanning tree (i.e., the spanning tree of minimum weight). In the end of the process a node outputs this weight.
- Given two nodes s and t, the **shortest s-t path** problem is to find the shortest path between s and t. In the end of the process a node outputs the length of the shortest path.
- Given a node s, the **s-source distance** problem [53] is to find the distance from s to every node. In the end of the process, every node knows its distance from s. The **s-source shortest path tree** problem [55] is to find the shortest path spanning tree rooted at s, i.e., the shortest path from s to any node t has the same weight as the unique path from s to t in such a tree.
- A set of edges E' is a **cut** if G is not connected when we delete E'. The **minimum cut** problem [52] is to find a cut of minimum weight. A set of edges E' is a s-t cut if s and t are not connected when we delete E'. The **minimum s-t cut** problem is to find an s-t cut of minimum weight. In the end of the process, a node outputs the weight of the minimum cut and minimum s-t cut. The **maximum cut** problem is to find a cut of maximum weight.
- The **minimum routing cost spanning tree** problem [150] is defined as follows. We think of the weight on an edge as the cost of routing messages through

this edge. The routing cost for a pair of vertices in a given spanning tree is the sum of the weights of the edges in the unique tree path between them. The routing cost of the tree itself is the sum over all pairs of vertices of the routing cost for the pair in the tree. Our goal is to find a spanning tree with minimum routing cost.

- The **generalized Steiner forest** problem [86] is defined as follows. We are given k disjoint subsets of vertices $V_1, ..., V_k$. The goal is to find a minimum weight subgraph in which each pair of vertices belonging to the same subsets are connected.

- Given a network with two cost functions associated to edges: weight and length. Given a root node r and the desired radius ℓ, a **shallow-light tree** [128] is the spanning tree whose radius (defined by length) is at most ℓ and the total weight is minimized (among trees of the desired radius).

We recall the following standard notion of an approximation algorithm which we defined earlier in Section 5.4. For any minimization problem $\mathcal{X}$, we say that an algorithm $\mathcal{A}$ is an α-approximation if, for any input instance $\mathcal{I}$, algorithm $\mathcal{A}$ outputs a solution that is at most α times the optimal solution of $\mathcal{I}$.

Therefore, in the minimum spanning tree, minimum cut, minimum s-t cut, and shortest s-t path problems stated above, an α-approximation algorithm should find a solution that has total weight at most α times the weight of the optimal solution. For the s-source distance problem, an α-approximation algorithm should find an approximate distance $d(v)$ of every vertex v such that $\mathit{distance}(s, v) \leq d(v) \leq \alpha \cdot \mathit{distance}(s, v)$ where $\mathit{distance}(s, v)$ is the distance of s from v. Similarly, an α-approximation algorithm for s-source shortest path tree should find a spanning tree T such that, for any node v, the length ℓ of a unique path from s to v in T satisfies $\ell \leq \alpha \cdot \mathit{distance}(s, v)$.

Additionally, we say that a randomized algorithm $\mathcal{A}$ is α-approximation ϵ-error if, for any input instance $\mathcal{I}$, algorithm $\mathcal{A}$ outputs a solution that is at most α times the optimal solution of $\mathcal{I}$ with probability at least $1 - \epsilon$.

Proof of Theorem 5.6. The proof idea for these problems is similar to the proof that the general case Traveling Salesman Problem cannot be approximated within $\alpha(n)$ for any polynomial computable function $\alpha(n)$ (see, e.g., [144]): We will define a weighted graph G' in such a way that if the subgraph H satisfies the desired property then the approximation algorithm must return some value that is at most $f(n)$, for some function f. Conversely, if H does not satisfy the property, the approximation algorithm will outputs some value that is strictly more than $f(n)$. Thus, we can distinguish between the two cases. We have already used this technique to proof the lower bound of the MST problem in Section 5.4. We now show lower bounds of other problems using the same technique.

The lower bound for shallow-light tree follows immediately when we set the length of every edge to be one and radius requirement to be n. In this case, the spanning tree satisfies the radius requirement and so the minimum-weight shallow-light tree becomes the minimum spanning tree.

The lower bound of s-source distance and shortest path spanning tree follow in a similar way: H is a spanning connected subgraph if and only if the distance from s to every node is at most $n - 1$ (i.e., $\mathcal{A}$ have approximate distance at most $(n-1)\alpha(n)$) if and only if the shortest path spanning tree contain only edges of weight one (i.e., the total weight of the shortest path spanning tree is at most $(n-1)\alpha(n)$).

For the lower bound of the shortest s-t path, observe that s and t are connected in H if and only if the distance from s to t in G' is at most $n - 1$, i.e., $\mathcal{A}$ outputs a value of at most $(n-1)\alpha(n)$. The lower bound follows from the lower bound of s-t connectivity verification problem.

For the lower bound of the minimum cut, first observe that H is a spanning connected component if and only if $\bar{H}$, obtained by deleting all edges $E(H)$ of H from G, is not a cut. (Recall that G is assumed to be connected in the problem definition.) Therefore, verifying if $\bar{H}$ is a cut also has the same lower bound. Now, we define $\bar{G}'$ by assigning weight one to all edges in $\bar{H}$ and $n\alpha(n)$ to all other edges and use the fact that $\bar{H}$ is a cut if and only if $\bar{G}'$ has minimum cut of weight at most $n-1$, i.e., $\mathcal{A}$ outputs value at most $(n-1)\alpha(n)$. The same argument applies to s-t cut: s and t are *not* connected in H if and only if $\bar{H}$ is an s-t cut if and only if $\bar{G}'$ has minimum $s-t$ cut of weight $n-1$.

For the minimum routing cost spanning tree problem, we assign weight 1 to edges in H and $n^3\alpha(n)$ to other edges. Observe that if H is a spanning connected subgraph, the routing cost between any pair will be at most $n-1$ and thus the cost of the $\alpha(n)$-approximation minimum routing cost spanning tree will be at most $(n-1)\binom{n}{2}\alpha(n) < n^3\alpha(n)$. Conversely, if H is not a spanning connected subgraph, some pair of nodes will have routing cost at least $n^3\alpha(n)$ and thus the minimum routing cost spanning tree will cost at least $n^3\alpha(n)$.

For the generalized Steiner forest problem, we will reduce from the lower bound of s-r connectivity. We will have only one set $V_1 = \{s, r\}$. We assign weight 1 to edges in H and $n\alpha(n)$ to other edges. Observe that the minimum generalized Steiner forest will have weight at most $n-1$ if H is s-t connected and at least $n\alpha(n)$ otherwise. (Recall that G is assumed to be connected in the problem definition.) □

Our MST lower bound here matches the lower bound of exact MST algorithms and improves the lower bound of $\Omega(\sqrt{\frac{n}{\alpha B}})$ by Elkin [55]. Our lower bound for s-source distance complements the results in [53].

5.8 Tightness of lower bounds

We note that all lower bounds of verification problems stated so far are almost tight. To show this we will present deterministic $O(\sqrt{n}\log^* n+D)$-time algorithms for the s-t connectivity, k-component, connectivity, cut, s-t-cut, bipartiteness, edge on all path, and simple path verification problems. Algorithms for all other problems stated in this chapter can be found using the reductions given in Figure 11.

In particular, one can use the MST algorithm by Kutten and Peleg [96] and the connected component algorithm by Thurimella [142][Algorithm 5] to verify these properties.

Deterministic algorithms almost matching the deterministic lower bounds: We need to give upper bounds for the k-spanning tree and path verification problems.

Path verification problem: compute a breath first search-tree T on $G \setminus H$ in time $O(D)$ connect the tree T to H by a single edge of G. The resulting subgraph of G is a spanning tree of G if and only if H is a path.

k-spanning tree verification problem: We construct a weighted graph G' by assigning weight zero to all edges in $H^k := H$ and one to other edges. We then find a minimum spanning T^k tree of H using the $O(\sqrt{n}\log^* n + D)$-time algorithm in [96]. Now we create $H^{k-1} := H^k \setminus T^k$. H^{k-1} is a $k-1$-spanning tree if and only if H^k is a k-spanning tree. If all T^j were spanning trees, after k iterations we are left with H^0 which contains no nodes and no edges if and only if H^k was a k-spanning tree.

Deterministic algorithms almost matching the randomized lower bounds: We need to give upper bounds for the s-t connectivity, cycle, connectivity, k-components, cut, s-t cut, bipartiteness and edge on path-verification problems.

s-t connectivity verification problem: To do this, we run the connected component algorithm by Thurimella [142][Algorithm 5] where, given a subgraph H of G,

the algorithm outputs a label $\ell(v)$ for each node v such that for any two nodes u and v, $\ell(u) = \ell(v)$ if and only if u and v are in the same connected component. [142][Theorem 6] states that the distributed time complexity of [142][Algorithm 5] is $O(D + f(n) + g(n) + \sqrt{n})$ where $f(n)$ and $g(n)$ are the distributed time complexities of finding an MST and a $\sqrt{n}$ -dominating set, respectively. Due to [96] we have that $f(n) = g(n) = O(F + \sqrt{n}\log^* n)$. We can now verify whether s and t are in the same connected component by verifying whether $\ell(s) = \ell(t)$.

cycle verification problem: Assign weight 0 to all edges of H and weight 1 to all edges of $G \setminus H$. Compute a minimum spanning tree of G using [96]. H contains no cycle if and only if all edges E_H of H are in the minimum spanning tree, i.e., $W = n - 1 - |E(H)|$ where $|E(H)|$ is the number of edges in H.

edge on all path verification problem: If and only if u and v are disconnected in $H \setminus \{e\}$, then e is on all paths between u and v. We can use the s-t connectivity verification algorithm from above to check that.

cut verification problem: To verify if H is a cut, we simply verify if G after removing the edges E_H of H is connected.

s-t cut verification problem: To verify if H is an s-t cut, we simply verify s-t connectivity of G after removing the edges E_H of H.

e-cycle verification problem: To verify if e is in some cycle of H, we simply verify s-t connectivity of $H' = H \setminus \{e\}$ where s and t are the end nodes of e. It is thus left to verify s-t connectivity.

k-components verification problem: We simply put weight 1 on edges in H and 2 on other edges and find the MST using an algorithm in [96]. Observe that H has at most k connected component if and only if there are at most $k - 1$ edges of weight 2 in the MST, i.e., the MST has weight at most $n - 1 + (k - 1)$.

connectivity verification problem: Same as above for $k = 1$.

spanning connected subgraph verification problem: One can use the above described algorithm to verify that H is connected. Verifying that the vertices V_G of G are the same as the vertices V_H of H completes the algorithm.

bipartiteness verification problem: Compute a minimum spanning tree T_H of H in time $O(\sqrt{n}\log^* n + D)$ using [96]. Now 2-color this tree. Now all nodes check if their neighbors have a different color then themselves. They will have a different color if and only if H is bipartite.

5.9 Conclusions

We initiate the systematic study of verification problems in the context of distributed network algorithms and present a uniform lower bound for several problems. We also show how these verification bounds can be used to obtain lower bounds on exact and approximation algorithms for many problems.

In light of the success in proving distributed algorithm lower bounds from communication complexity in this and the previous chapter, it is interesting to explore further applications of this technique. One interesting approach is to show a connection between distributed algorithm lower bounds and other models of communication complexity, such as multiparty and asymmetric communication complexity (see, e.g., [95]). Potential applications of this technique are distance- and cut-related problems such as shortest s-t path, single-source distance computation, shortest path tree, s-t cut, and minimum cut. (Some of these problems were also asked in [52].) The lower bound of $\Omega(\sqrt{n})$ are shown in this chapter for these types of problems and stronger lower bounds such as $\Omega(\sqrt{nD})$ or $\Omega(n)$ are still possible. Another general direction for extending all of this work is to study similar verification problems in special classes of graphs, e.g., a complete graph.

Related publications. The preliminary version of this chapter appeared as a joint work with Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Gopal Pandurangan, David Peleg, and Roger Wattenhofer [41].

CHAPTER VI

SKYLINE COMPUTATION ON MULTI-PASS STREAMS AND DISTRIBUTED NETWORKS

Recall that the skyline of a d-dimensional dataset is the set of points (tuples) that are not *dominated* by any other point, where we say that a point p dominates another point p' if the coordinate of p on each dimension is not smaller than that of p', and strictly larger on at least one dimension.

In this chapter, we consider computing skylines when the input cannot be stored in the main memory. As we discussed in Section 1.1, in this type of memory, the sequential disk access is preferable to the random disk access. We use the multi-pass streaming model (implicitly with physically sequential accesses of disks) for this problem. We are interested in worst case analysis, in terms of random and sequential I/O's and comparison operations. We prove a simple yet instructive performance lower bound under this model.

Recall that, in the multi-pass streaming model, the data is streamed sequentially through internal memory. It is possible that the data is streamed multiple times; we refer to each scan through the dataset as a pass. Additionally, we allow our algorithms to produce a new stream while they are reading the current stream data. This is a common method used in many database algorithms: Read data from one file and write a new file if necessary. We note that this can be implemented with only sequential disk access when there are at least two disks available. Such a model has recently been defined as a Read/Write streaming model [135, 136] and has received considerable attention from the theory community.

Organization: The key result of this chapter is a randomized multi-pass streaming algorithm, RAND. We present two versions, one with and one without fixed window, and prove their theoretical worst-case performance guarantees. Both versions can be found in Section 6.1). We then show a matching lower bound in Section 6.2. Then, in Section 6.3 we show extensions of this algorithm into a two-dimensional deterministic algorithm and an algorithm on the partially-ordered domains. We show through extensive experiments that RAND is comparable to the state-of-the-art skyline algorithms in various performance metrics. We also show that, with certain perturbations of the data orders, the performance of the other algorithms degrade considerably while RAND is robust to such changes. The experimental results can be found in Section 6.4. Finally, in Section 6.5 we extend the algorithm to an almost optimal distributed algorithm.

6.1 Streaming algorithms

In the rest of this chapter, we use n for the number of points, m for the number of skyline points, and d for the number of dimensions of each point. We measure the performance of algorithms in terms of random I/O's, sequential I/O's, and comparisons performed. The random I/O's is simply the number of passes performed by an algorithm in the streaming model. In this section, we present randomized algorithms for the following different settings:

- Streaming-RAND algorithm: This algorithm gives an efficient tradeoff between the memory space and the number of passes (random I/O's). In the worst case, it uses $O(m \log n)$ space, $O(\log n)$ passes (random I/O's), $O(n \log m)$ sequential I/O's and $O(dmn \log m \log n)$ comparisons with high probability.

- FixedWindow-RAND algorithm: This algorithm runs using a fixed memory space; i.e., for a predetermined window size w, it always stores at most w points. We use this algorithm to compare performance with previous algorithms in the

literature in Section 6.4. With window size w, this algorithm uses $O(m \log n/w)$ random I/O's, $O(mn/w)$ sequential I/O's and $O(dmn)$ comparisons in expectation. The high probability bounds are only $O(\log n)$ more than the expected bounds presented above. We also present them in this section.

Notice that the worst case is over all inputs, and the randomization (and therefore high probability or expectation bounds) is for the algorithms' coin tosses. In other words, our algorithms have the guarantees for *any* kind of input.

Outline The main idea is to quickly find skyline points that dominate many points in a few of passes. In particular, we present an algorithm that finds a set of skyline points which dominate about *half* of the input points in three passes, using memory space $O(m)$. The main idea is that such skyline points are easy to find by sampling: If we pick one input point uniformly at random and find a skyline point that dominates this point, then we are likely to find a skyline point that dominates more points than another. The main goal of our sampling technique is to be able to sample skyline points. However, there are two difficulties: it is not obvious how to sample a skyline point from all skyline points (as these are not known to the algorithm). Further, sampling skyline points uniformly does not necessarily ensure dominating a lot of points. We wish to sample skyline points in proportion to the number of points they dominate. It turns out that sampling roughly $24m$ points is sufficient to find skyline points that dominate at least $\frac{3n}{4}$ points, in expectation. We now describe the algorithms and their analysis in details.

6.1.1 Key idea: Eliminating points in a few passes

We start with the following simple question: If m is known, how many points can we dominate using a constant-pass streaming algorithm with about m space? In this section we answer this question. We present a simple three-pass, $24m$-space

algorithm, ELIMINATE-POINTS (cf. Algorithm 6.1.1), that guarantees to eliminate at least $\frac{3n}{4}$ elements in expectation. The tricky part here is analyzing its performance. We later extend this algorithm to other cases.

Algorithm 6.1 ELIMINATE-POINTS (m)

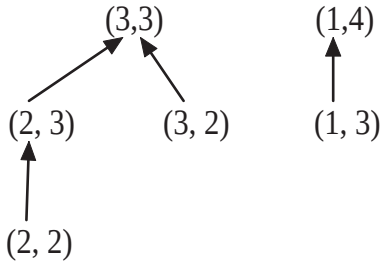
Input: $p_1, p_2, ..., p_{n'}$ (in order) where n' is the number of points in the stream.
Output: Skyline points S'

1: Let $x = 24m$.
2: **Pass 1:** For $j = 1, 2, ..., x$, let p'_j be a point picked uniformly at random from the stream. Let S be the set of such points.
3: **Pass 2:**
4: **for** $i = 1..n'$ **do**
5: For any p'_j, if p_i dominates p'_j then $p'_j := p_i$
6: **end for**
7: Let $S' = \{p'_1, p'_2, ...p'_x\}$.
8: **Pass 3:** Delete from stream all points in S' and all points dominated by any point in S'.
9: **return** S'

The main idea of the ELIMINATE-POINTS algorithm is to sample $24m$ points from the stream. (This sampling can be done in one pass using *reservoir sampling* [145].) Next, we spend one more pass to replace these sampled points by points that dominate them. We note that it is important to do this in order (as it is crucial in the analysis of Lemma 6.1). Points obtained at the end of this pass are skyline points (although not necessarily distinct). In the last pass, we delete points we obtain from the second pass and any points dominated by them.

Now we analyze the algorithm. First, we analyze the performance in expectation. To simplify the analysis, we assume that the sampling is done *with replacement*; that is one point may be sampled more than once. The performance of the algorithm will be better if we sample without replacement.

Lemma 6.1. *After* ELIMINATE-POINTS *algorithm with parameter m, the expected number of points left in the stream is at most $n'/4$, where n' is the number of points in the input stream.*



Input order: (1,4), (2,3), (1,3), (3,2), (3,3), (2,2)

Figure 18: Example from the proof of Lemma 6.1.

Proof. First, we construct the following directed graph, denoted by G. The goal here is to allocate each point to a unique skyline point that dominates it (note that there could be multiple skyline points that dominate a given point). This graph consists of n' vertices corresponding to points in the stream. We abuse notation and denote the vertices by $p_1, p_2, ..., p_{n'}$. It will be clear from the context whether we refer p_i as a point in the stream or a vertex. For each i, we draw an edge from p_i to p_j if and only if p_j is the first point (leftmost) in the stream that dominates p_i. Figure 18 shows an example. (In the example, notice that $(2, 2)$ points to $(2, 3)$ but not $(3, 2)$ since $(2, 3)$ is the first point in the stream that dominates $(2, 2)$. Also note that if $(2, 2)$ is sampled then it will be replaced by $(2, 3)$ and $(2, 3)$ will be later replaced by $(3, 3)$. Therefore, we get $(3, 3)$ as a product.)

Now, let $q_1, q_2, ..., q_m$ be the skyline points. For $i = 1, 2, ..., m$, define $S_i = \{p_j \mid \text{There is a path from } p_j \text{ to } q_i\}$. (These sets are disjoint and their union is the set of all points in the stream.)

Claim 6.2. *If a point p is sampled in Pass 1 then the skyline point q_i is in S' after the iteration ends where i is such that $p \in S_i$.*

Proof. Let $p = x_0, x_1, x_2, ..., x_t = q_i$ be the path from p to q_i. By definition, p is replaced by x_1. Observe that x_2 appears after x_1 in the stream (since x_2 also

dominates p). Therefore, x_1 is replaced by x_2. For the same reason, we conclude that x_j is replaced by x_{j+1} for all $j = 0, 1, ..., t$. In the end, q_i is in S as claimed. □

The main observation to make is that the sampled point p is in fact replaced, one by one as the stream progresses, by points on the path from p to q_i.

By Claim 6.2, a point will remain in the stream after the algorithm finishes only if none of the points in the set S_i containing it are picked in Pass 1. In other words, the number of points left in the stream is at most $\sum_{i:\ S_i \cap S = \emptyset} |S_i|$ (recall that S is the set of points sampled in Pass 1). We now bound the expected value of this quantity.

Claim 6.3. $\mathbb{E}[\sum_{i:\ S_i \cap S = \emptyset} |S_i|] \leq (n'/4)$.

First, we mention the intuition behind the proof. First, notice that the bigger S_i is, the more likely a point in S_i will be sampled. Consequently, it is more likely that we end up with a skyline point q_i as compared to q_j if $|S_i| > |S_j|$. This is the key insight in the algorithm, as the random sampling biases the algorithm towards skyline points that are likely to dominate a larger number of points.

However, can we argue that at most $\frac{n'}{4}$ are left? Consider all S_i such that $|S_i| \geq \frac{n'}{8m}$, i.e., the *large* S_i's. For any of these large sets, its size is $1/8m$ fraction of all the points. Therefore, sampling at least $8m$ points will get us at least one point in this set in expectation. However, we may not find samples from all large sets. A clever analysis can be used to show that we eliminate a large fraction of points in these large sets. Finally, a counting argument shows that the total number of points in these sets combined is a large fraction of all the points (to be precise, it is at least $\frac{7n'}{8}$). This completes the intuition of the proof.

of Claim 6.3. Without loss of generality, assume that $|S_1| \geq |S_2| \geq ... \geq |S_m|$ (by permuting). For any set S_i, the probability that no elements in the set S_i are picked

is $(1 - |S_i|/n')^{-24m} \leq e^{-24m|S_i|/n'}$. Therefore,

$$\mathbb{E}[\sum_{i:\ S_i \cap S = \emptyset} |S_i|] = \sum_{i=1}^{m} |S_i| e^{-24m|S_i|/n'}.$$

by union bound.

Now we break the quantity above into two terms with the sets of size at least and at most $n/(8m)$, respectively. That is, consider $S_{k+1}, S_{k+2}, ..., S_m$ where k is such that $|S_k| \geq n'/(8m)$ and either $k = m$ or $|S_{k+1}| < n'/(8m)$. We rewrite the previous quantity as

$$\sum_{i=1}^{k} |S_i| e^{-24m|S_i|/n'} + \sum_{i=k+1}^{m} |S_i| e^{-24m|S_i|/n'}.$$

We now bound the first term. When $|S_i| \geq n'/(8m)$,

$$e^{-24m|S_i|/n'} \leq e^{-3} \leq 1/(8).$$

Therefore, the first term becomes

$$\sum_{i=1}^{k} |S_i| e^{-24m|S_i|/n'} \leq \frac{1}{8} \sum_{i=1}^{k} |S_i| \leq \frac{n'}{8}.$$

For the second term, note that $|S_i| e^{-24m|S_i|/n'} \leq |S_i| \leq n'/(8m)$. Therefore, the second term becomes

$$\sum_{i=k+1}^{m} |S_i| e^{-24m|S_i|/n'} \leq \sum_{i=k+1}^{m} |S_i| \leq m \frac{n'}{8m} \leq \frac{n'}{8}.$$

Summing the two terms together, give the claimed bound $\mathbb{E}[\sum_{i:\ S_i \cap S = \emptyset} |S_i|] \leq n'/4$. □

The lemma follows immediately from Claim 6.3. □

6.1.2 Streaming algorithm

Now we develop a time/space-efficient streaming algorithm for finding all skyline points. We first give a high level idea of the algorithm. Let us focus on the number of passes for now. The basic idea is to apply the ELIMINATE-POINTS algorithm

repeatedly until no points are left in the stream. If m (the number of skyline points) is known ahead of time then this process is likely to finish in $O(\log n)$ steps, where n is the number of input points. This is because we are likely to delete half of the points in each application of ELIMINATE-POINTS.

However, the difficulty is that m is not known. One way to get over this is to use the standard doubling trick to find m: Start with $m' = 1$ (as the *guess* for the number of skyline points), run ELIMINATE-POINTS for $O(\log n)$ steps assuming that $m = m'$. If the problem is not solved, double the size of m' and repeat. Since we will be done (with high probability) when $m' \geq m$ we have to repeat only $\log m$ steps (with high probability). Therefore, this algorithm is likely to stop in $O(\log n \log m)$.

To get the number of passes to $O(\log n + \log m)$, we exploit Lemma 6.4 further. By such Lemma, if $m' \geq m$ then we are likely to eliminate half of the points in every pass. Therefore, if we find out that the algorithm eliminates less than such expected fraction, we double m' immediately instead of waiting for another $\log n$ passes. The algorithm is stated as the STREAMING-RAND algorithm (cf. Algorithm 6.2).

Algorithm 6.2 STREAMING-RAND

1: Let n be the number of points in the input stream. Let $m' = 1$.
2: **while** the input stream is not empty **do**
3: Let n' be the current number of points in the stream
4: Call ELIMINATE-POINTS($m' \log(n \log n)$)
5: If more than $n'/2$ points are left in the stream, $m' = 2m'$.
6: **end while**

Remark: In case the stream cannot be changed, we do not have to actually delete points from stream. We only keep the skyline points found so far and consider only points in the stream that is not dominated by any found skyline points.

We now give a formal analysis of this algorithm. The efficiency of the algorithm relies on the fact that we are likely to stop before m' gets too big, as shown in the following two lemmas.

Lemma 6.4. *After* ELIMINATE-POINTS *algorithm with parameter m, at most $n'/2$ points are left in the stream with probability at least $1/2$, where n' is the number of*

points in the input stream.

Proof. Let X be the number of point left in the stream after ELIMINATE-POINTS algorithm. Recall from Lemma 6.1 that $\mathbb{E}[X] \leq \frac{n'}{4}$. By the Markov's inequality, $Pr[X \geq \frac{n'}{2}] \geq Pr[X \geq 2\mathbb{E}[X]] \leq 1/2$. □

Lemma 6.5. *The probability that the algorithm repeats until $m' \geq 2m$ is at most $1/n$.*

Proof. Let τ be the first iteration that m' is at least m; that is, $m \leq m' < 2m$. Consider any iteration after τ. Recall that by Lemma 6.4, each run of ELIMINATE-POINTS(m), which samples $24m$ points, halves the stream with probability at least $1/2$. It follows that each run of ELIMINATE-POINTS($m \log(n \log n)$), which samples $24m \log(n \log n)$ points, halves the stream with probability at least $1 - 1/(n \log n)$. By union bound, the probability that all $\log n$ iterations after τ delete at least half the points is at least $1 - 1/n$. This implies that, with probability $1 - 1/n$, the stream will be empty before m' is increased again. □

Now we analyze the algorithm in all aspects.

Theorem 6.6. STREAMING-RAND *algorithm (cf. Algorithm 6.2) uses with probability at least $1 - 1/n$,*

1. *$O(m \log n)$ space*
2. *$O(\log n)$ random I/O's (passes)*
3. *$O(n \log m)$ sequential I/O's, and*
4. *$O(dmn \log n \log m)$ comparisons.*

Proof. By Lemma 6.5, $m' < 2m$ with probability at least $1 - 1/n$. We show that the theorem holds when this happens.

For the first claim, when $m' < 2m$ the space becomes $O(m \log(mn \log n)) = O(m \log n)$ as claimed.

For the second claim, observe that the algorithm in each iteration either scales m' up twice or scales n' down by half. It can scale down n' for only $\lceil \log n \rceil$ times and it scales m' up for only $\lceil \log m \rceil$ times (before $m' \geq 2m$). We thus prove the second claim.

For the third claim, first we count the number of sequential I/O's made by iterations that increases m'. There are $\lceil \log m \rceil$ such iterations (if $m' < 2m$) and each iteration reads through the stream three times. So, the total number of sequential I/O's used by these iterations is $O(n \log m)$. For the number of sequential I/O's used by the remaining iterations, observe that the size of the stream is at most $n/2^{i-1}$ in the i-th such iteration (because such iterations scale the size of the stream down by half). Therefore, the number of sequential I/O's for this m' is $O((1 + 1/2 + 1/2^2 + ...)n) = O(n)$. We thus prove the third claim.

For the last claim, observe that when the algorithm reads a new element from the stream, it compares this element with all elements in the memory; i.e., it compares $O(m \log n)$ pairs of points per sequential I/O. Further, comparison of any pair of points requires comparing d different dimensions in worst case. The claim thus follows from the third claim that there are $O(n \log m)$ sequential I/O's. □

Notice that this is a near-optimal trade-off between space and passes of any streaming algorithm, since our lower bound shows that with one pass, any algorithm requires $\Omega(n)$ space. We increases the passes by only a logarithmic factor and get almost optimal space bounds.

6.1.3 Fixed-window algorithm

Many of the previous skyline algorithms are presented with predetermined window size w. That is, the algorithm is allowed to store only $O(w)$ points in the stream; this

might be a constrained due to the memory specifications of the machine. The goal is to analyze a variant of the algorithm for the numbers of random I/O's (passes), sequential I/O's, and comparisons under this setting. We show that the following very simple algorithm is efficient:

FixedWindow-RAND: While the stream is not empty, call Eliminate-Points($\lfloor w/24 \rfloor$).

Now we analyze the algorithm. We first state the theorem in terms of expectation, and then the high probability bound.

Theorem 6.7. *In expectation,* FixedWindow-RAND *algorithm uses* $O(\frac{m}{w} \log n)$ *random I/O's,* $O(\frac{nm}{w})$ *sequential I/O's, and* $O(dnm)$ *comparisons.*

Proof. Consider running Eliminate-Points($\lfloor w/24 \rfloor$) for $\lceil 24m/w \rceil$ times. We claim that this process is as efficient as running Eliminate-Points(m). (In other words, the distribution of the running time of the former process stochastically dominates that of the latter one.) Intuitively, this is because both processes get the same number of samples but the former process does not sample points that are dominated by the previous samples. We now analyze the process of running Eliminate-Points(m).

The key idea is to consider the runs of Eliminate-Points(m) that reduces the stream by at least half. Let us call these runs "success" runs and the rest runs "fail". Recall from Lemma 6.4 that each run succeeds with probability at least 1/2.

For the expected number of times we have to run Eliminate-Points(m), observe that we have to run the algorithm until we see $\log n$ success runs. Since each run succeeds with probability 1/2, the expected number of times we have to run Eliminate-Points(m) is at most $2 \log n$. Multiplying this number by $\lceil 24m/w \rceil$ gives the number of random I/O's claimed in the theorem statement.

For sequential I/O's, we claim that the expected number of fails between each pair of consecutive successes is 1. To be precise, for $i = 0, 1, ..., \log n$, let X_i be the

number of fail runs between the i-th and $(i+1)$-th success runs. Since each run succeeds with probability at least $1/2$, $\mathbb{E}[X_i] \leq 1$ for all i, as claimed. Now, observe that the number of sequential I/O's is at most $2n(X_0 + X_1/2 + X_2/4 + ...)$ since each success run halves the stream. Therefore, by the linearity of expectation, the expected number of sequential I/O's is $2n(\mathbb{E}[X_0] + \mathbb{E}[X_1]/2 + \mathbb{E}[X_2]/4 + ...) \leq 4n$. The expected number of sequential I/O's claimed in the theorem statement follows by multiplying the number by $\lceil 24m/w \rceil$.

For the number of comparisons, observe that we compare each element read from the stream with w elements in the window. Moreover, the number of elements read from the stream is bounded by the number of sequential I/O's which is $O(nm/w)$. Therefore, there are $O(nm)$ comparisons. Each vector comparison needs d comparisons. The theorem is thus completed. □

Theorem 6.8. *With high probability,* FIXEDWINDOW-RAND *algorithm uses* $O(\frac{m \log n}{w})$ *random I/O's,* $O(\frac{nm \log n}{w})$ *sequential I/O's, and* $O(dnm \log n)$ *comparisons.*

Proof. The random I/O's can be proven by Chernoff bounds (essentially arguing that in $2m \log n/w$ executions with w space with high probability, at least $m \log n/w$ executions result in eliminating half the points). Observe that in $O((m \ln(mn \log n))/w)$ passes we get samples equal to one round of ELIMINATE-POINTS. The lemma follows immediately from the previous section. An important point to note is that, in expectation, the number of sequential I/Os (and consequently comparisons) save a $\log n$ factor as there is no need to perform the doubling trick. □

6.1.4 Algorithm Comparisons

We compare performance of the FIXEDWINDOW-RAND algorithm (referred to as RAND from now on) in the worst case against BNL and LESS, two previously proposed non-preprocessing fixed-window algorithms. The asymptotic performance bounds are summarized in Table 2. Recall that n denotes the number of input points,

Table 2: Comparison of algorithms

Algorithm	Random I/O's	Sequential I/O's	comparisons
BNL(w)	$\Theta(\min\{m, \frac{n}{w}\})$	$\Theta(\min\{mn, \frac{n^2}{w}\})$	$\Theta(d \min(wmn, n^2))$
LESS(w)	$\Theta(n \log_w \frac{n}{w})$	$\Theta(\frac{mn}{w})$	$\Theta(dmn + n \log n)$
RAND(w)	$O(\frac{m \log n}{w})$	$O(\frac{mn}{w})$	$O(dmn)$

m denotes the number of points in the skyline, and d denotes the dimension.

We analyze three parameters that affect the algorithms' performance: random I/O's, sequential I/O's, and the number of comparisons. Although, intuitively, random I/O's are those I/O's that need seek operations, this definition could be confusing sometimes as sequential I/O's can be sometimes counted as random I/O's. To avoid this confusion, we define a random I/O to be I/O that reverses the hard disk head: the previously read element appears after the newly read element on the disk.

The expected bound of RAND is shown in Theorem 6.7. For BNL and LESS, the upper bounds are already shown many times in the literature and the the formal proof of the lower bounds can be constructed by creating fairly simple tight examples. We omit them here for brevity.

As can be seen from the comparisons table, in all the algorithms, the number of sequential I/O's is significantly more than the number of random I/O's. However, the cost of a random I/O can be a lot more than the cost of a sequential I/O. Therefore, it is important to analyze these separately. In terms of random I/O's, LESS scales linearly with n, so is worse than BNL and RAND. Further, RAND is better than BNL with a saving of almost a w factor (the random I/O's of RAND are actually the minimum of what is written in the table for BNL and for RAND). In terms of sequential I/O's, LESS and RAND are asymptotically the same, and better than BNL by a w-factor.

Apart from I/O's, comparisons made by algorithms is a key contributor to the overall performance. LESS has a sorting phase because of which it incurs $\Theta(n \log n)$

more comparisons than RAND. Further, BNL incurs roughly a w-factor extra comparison cost as compared to RAND (which is why BNL's cost increases when the window size is increased beyond a point).

These interpretations are corroborated by the results in our evaluations section.

6.2 An almost tight lower bound of streaming skyline algorithms

We have shown an algorithm that finds skyline using $O(m \operatorname{polylog} n)$ space and $O(\log n)$ passes. In this section, we show that if we want to maintain the space to be $O(m \operatorname{polylog} n)$ then the number of passes is almost tight, as in the following theorem.

Theorem 6.9. *Any streaming algorithm that uses $O(m \operatorname{polylog} n)$ space requires $\Omega(\log n / \log \log n)$ passes.*

To prove this theorem, we consider a variant of the pointer chasing problem (cf. Section 4.1) where Alice and Bob receive d pairs of functions (f_A^1, f_B^1), ..., (f_A^d, f_B^d), where each function maps from $[t]$ to $[t]$. They want to output d values resulting from chasing these pairs of functions for r rounds each. We denote this problem by $\text{PC}^{r,t,d}$. That is, for any i, we want to compute $g_k^i = f_B^i(f_A^i(...f_B^i(f_A(1))...))$ where we see f_A and f_B in the expression for k times each. Jain, Radhakrishnan, and Sen [78] show that if Alice and Bob want to compute $\text{PC}^{r,t,d}$ in less than r rounds, they cannot do anything much better than naively applying each function separately for $r+1$ times, as in the following theorem [1].

Theorem 6.10. *[78] For any $r' < r$, $R_{1/3}^{r'-cc-pub}(\text{PC}^{r,t,d}) = \Omega(dtr^{-3} - dr \log t - 2d)$.*

Using the above result, we show the following lemma which will be used to show Theorem 6.9.

[1] In fact this holds even when Alice and Bob are allowed r rounds but Alice cannot send any message in the first round

Lemma 6.11. *For any d, t, and r, there exists a set of two-dimensional points of size $n = O(dt^r)$ with skyline size $m = 2dr + 2$ such that the space lower bound of any r-pass skyline algorithm is $\Omega(dt/r^4)$.*

Proof. For each $d' \in [d]$, we put the following t^r points $x^{d'}_{11\ldots11}, x^{d'}_{11\ldots12}, \ldots, x^{d'}_{tt\ldots tt}$ to the stream. Each point $x^{d'}_{i_1 i_2 \ldots i_r}$ will be used to represent the event that we start chasing a pointer using functions $f^{d'}_A$ and $f^{d'}_B$ from i_1 then go to i_2 then i_3 and so on. We call these points the x-points.

Additionally, for each input function $f^{d'}_A$, odd number j and $t' \in [t]$, Alice constructs a point $a^{d'}_{j,1}, a^{d'}_{j,2}$ that dominates all points except those in the form $a_{i_1,\ldots,i_{j-1},i_j,f_A(i_j),\ldots,i_r}$ where $i_j = t'$. Similarly, Bob constructs points $b^{d'}_{j,1}, b^{d'}_{j,2}$ for every even number j.

Observe that the only x-point that is not dominated by any points constructed by Alice and Bob is $x_{i_1 i_2 \ldots i_r}$ where $i_j = g^{d'}_j(1)$ for any j. Thus, if there is an r'-round streaming algorithm to compute the skyline then Alice and Bob can solve $\textsc{pc}^{r,t,d}$ by simulating such algorithm and looking at the x-point left in the skyline. Theorem 6.10 implies that if $r' < r$ then in some round Alice and Bob need to communicate $\Omega(dt/r^4)$ bits. This implies the lower bound of the space of streaming algorithms. □

Corollary 6.12. *For any s and constant c, any streaming algorithm that uses $O(s \log^c n)$ space requires $\Omega(\frac{\log n}{\log\log n} \times (\frac{m}{s})^{1/5})$ passes, for infinitely many values of n and m.*

Proof. We know that, to finish in less than k rounds, the space must be $s \log^c n = \Omega(dt/k^4)$ by the previous lemma. Using $n = O(dt^k)$, i.e. $dt = n^{1/k} d^{1-1/k}$, we get $s \log^c n = \Omega(n^{1/k} d^{1-1/k}/k^4)$. Using $m = dk$, we have $s \log^c n = \Omega(n^{1/k} m/k^{5-1/k})$. Therefore we get $k = \Omega(n^{1/k} m/(s \log^c n))^{1/5}$.

We note that $k \geq n^{1/(5k)/\log^{c/5} n}$ implies that $k \geq \log n/(5c \log\log n)$ since if $k < \log n/(5c \log\log n)$ then $n^{1/(5k)}/\log^{c/5} n > n^{c \log\log n/\log n}/\log^{c/5} n = \log^c n/\log^{c/5} n > k$. Thus, the number of passes is $k = \Omega(\frac{\log n}{\log\log n} \times (\frac{n'}{s})^{1/5})$ as desired. □

6.3 Extensions

In this section, we discuss about extending our algorithms to other settings, i.e., we show a deterministic variant of the algorithms for the case of 2-dimensions and that the algorithms can solve a more general problem on posets.

6.3.1 Deterministic 2D algorithm

Among deterministic and randomized algorithms with the same performance, the former is preferable. We show that when the points have only two dimensions, there is an efficient deterministic skyline streaming algorithm. We note that previous algorithm [94] could not be adapted for sequential access since it requires sorting.

The main idea is to replace the ELIMINATE-POINTS by the deterministic algorithm called 2D-ELIMINATE-POINTS (cf. Algorithm 6.3.1).

Algorithm 6.3 2D-ELIMINATE-POINTS

1: Let n be the size of the stream
2: **Pass 1:** Compute ϵ-approximate quantile summary of the first coordinate using Greenwald-Khanna's algorithm [70] where $\epsilon = 1/4m$. From this summary, find $v_0, v_1, ..., v_m$, which are the 0-th, $(1/m)$-th, $(2/m)$-th, ..., (m/m)-th approximate quantiles respectively.
3: **Pass 2:** For any $0 \leq i \leq m$, let p'_i be a point such that $v_i \leq p'_i[1] < v_{i+1}$. Let $S = \{p'_0, p'_1, ..., p'_m\}$
4: **Pass 3:** For any p_i: **(1)** If p_i is dominates some points in S then delete those dominated points from S. **(2)** If (1) holds and p_i is not dominated by any points in S then add p_i to S.
5: **Pass 4:** Delete all points in S and all points dominated by any point in S from the stream.
6: **return** S

The main component of the 2D-ELIMINATE-POINTS algorithm is *Greenwald-Khanna's approximate quantile summary* [70]: Suppose we read n numbers from stream $a_1, a_2, ..., a_n$. Let π be a sorting permutation of data; i.e., $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$. For any $0 \leq \phi \leq 1$, the approximate ϕ-quantile is a number t that is not too far from $a_{\lceil \phi n \rceil}$; i.e., $a_{\pi(\lceil \phi n - \epsilon n \rceil)} \leq t \leq a_{\pi(\lceil \phi n + \epsilon n \rceil)}$. In particular, the value of v_i has rank between $(ni)/m - \epsilon n = (ni)/m - n/(4m)$ and $(ni)/m + \epsilon n = (ni)/m + n/(4m)$.

As a consequence, for any i the number of element of value (of the first coordinate) between v_i and v_{i+1} is between $n/(2m)$ and $6n/(4m)$. (Note that v_0 is always the minimum value and v_m is the maximum value.)

Lemma 6.13. *After the* 2D-Eliminate-Points *algorithm finishes, either* $|S| \geq m/2$ *or the stream size is at most* $n/4$.

Proof. Let S' be the set returned by the algorithm. Suppose that $|S'| < m/2$. We partition the points $p'_0, p'_1, ..., p'_m$ obtained from Pass 2 into three sets. Let S_0 be the set of p'_i's that are in S after the algorithm ends. Let S_1 be the set of p'_i such that p'_i is dominated by a point in S_0, or there is a point p in S' that dominates p'_i and p'_j for any $j > i$. Let S_2 be the rest of the points. Note that every point in S_1 and S_2 must be dominated by some point in S'. We present two simple claims whose proofs are placed in the appendix.

Claim 6.14. $|S_1| > m/2$.

Proof. Draw a line from every point in S_0 to itself in S' and from every point in S_1 and S_2 to any point in S' that dominates them. Since those points in S_0 and S_2 share no points in S' with other points and $|S'| < m/2$, $|S_0| + |S_2| < m/2$. The claim thus follows. □

Claim 6.15. *For any* i, *if* $p'_i \in S_1$ *then every point* p *in the stream such that* $v_i \leq p[1] < v_{i+1}$ *is dominated by some point in* S'.

Proof. Observe that in both cases that makes p'_i in S_1, there is a point q and an integer $j > i$ such that $q[1] \geq v_j$. Moreover, since q dominates p'_i, $q[2] > p'_i[2]$. However, $p'_i[2] \geq p[2]$ for any point p such that $v_i \leq p[1] < v_{i+1}$ (by Pass 2). The claim follows. □

It follows, as before, that for any i there are at least $n/(2m)$ points whose first coordinate lies between v_i and v_{i+1}. Therefore, there are at least $|S_1| \cdot \frac{n}{2m} > n/4$ points deleted from the stream. □

We obtain deterministic algorithms by using 2D-Eliminate-Points as a subroutine of the Streaming-Skyline and FixedWindow-Streaming-Skyline algorithms.

6.3.2 Posets

Although the algorithms we described so far are developed primarily to compute skylines where the input is assumed to be a set of vectors, they can in fact solve more general problems on mathematical objects called *posets*.

We begin by defining posets formally. A partial order is a binary relation $\preceq$ that is reflexive, antisymmetric and transitive. (That is, for any element a, $a \preceq a$. For any a and b, if $a \preceq b$ and $b \preceq a$ then $a = b$. And, for any a, b and c, if $a \preceq b$ and $b \preceq c$ then $a \preceq c$.) A set with a partial order is called a *partially ordered set* (poset). It is easy to see that the following forms a poset: a set of vectors where we say that $p \preceq q$ if and only if p dominates q, for any vector p and q.

The problem of finding minimal elements on posts is to find all element p such that there is no element $q \preceq p$. In other words, we want to find all elements that are not "dominated" by any other elements. It follows that skyline computation is a special case of such problem when the elements are vectors and "$\preceq$" is equivalent to "dominate".

One of the results in Daskalakis et al. is a randomized algorithm that solves the problem using $O(\omega n + \omega^2 \log n)$ "queries" in expectation where ω is the "width" of the poset. Since our algorithms presented in section 6.1 do not need any structure of vectors, it can be used to find minimal elements on posets as well. In particular, the Streaming-Skyline is a streaming algorithm that uses $O(mn) = O(\omega n)$ queries in

Dataset	n	dimensions	Skyline size
House	127931	6	5774
NBA	17264	5	495
Island	63383	9	467
Color	68040	9	1533

Figure 19: Comparison of Real Datasets

expectation. Moreover, it uses only $O(m \log n)$ space, $O(\log n)$ passes. Thus, it is asymptotically as good as the previous algorithm in terms of the number of queries. Moreover, to the best of our knowledge, it is the first non-trivial streaming algorithm.

This version of our algorithm on posets can be adapted to compute skylines with partially-ordered domains (see, e.g., [28, 148, 149, 133] and references therein).

6.4 Experimental Evaluation

In this section we present results from an experimental study comparing our external randomized algorithm (FixedWindow-RAND or RAND) with the best known non-indexing algorithms. All our implementations were done in Java. To remove the effects of caching on our I/O times, we padded each record up to 104 bytes, similar to [21, 65]. Furthermore, we decreased the memory size of our machine to just 256MB so that very little memory would be available for caching. All our experiments were performed on a dual-core 1.7GHz Intel Xeon running Linux 2.6.9. The machine had two hard disks and we implemented all algorithms to use both of them.

We performed experiments on both synthetically generated and real datasets. We generated independent, correlated, and anti-correlated data using the dataset generator that was provided by the authors of [21]. The real data sets that we used are summarized in Table 19 [2]. For all these datasets, we computed the skyline using *min* in all dimensions.

[2]All these datasets are available for download from Dr. Yufei Tao's homepage: http://www.cse.cuhk.edu.hk/∼taoyf/. We note that Island is in fact a synthetic data set [139].

We compared the performance of our algorithm against the state of the art external non-indexed algorithms : BNL [21] and LESS [65]. In particular, we did not compare against indexed algorithms such as BBS [123] as they are not comparable in this setting. We also evaluated the performance of some other algorithms, including SFS [32] and BEST [143], but they were considerably slower (as noted earlier in [65]) and we do not show the results for them here.

In our experiments, we compare the wall-clock running time, number of comparisons and I/O's. Unless otherwise stated, each experiment used as default a window size equal to 1% of the number of points (as in [21, 65]). Therefore, in our implementation, for any comparison, all three algorithms use same amount of memory. The synthetic data sets each had one million data points and five dimensional data.

Since our algorithm is randomized we measured the variance in its performance for a single data set. We repeatedly ran RAND on the House data 500 times and measured the standard deviation of the running time to be less than 1% of the mean. Furthermore, in 95% of the runs the running time of RAND exceeded the mean running time by at most 2%, and in all the runs it never exceeded the mean by more than 4%. Hence, the performance of RAND does not vary much due to randomization.

6.4.1 Baseline Comparisons

We compared our algorithm with BNL and LESS for all the datasets described above. For brevity, though, we only show the results for the House and anti-correlated datasets, and comment on how the results were different in other cases.

We do not wish to assert any quantitative comparisons between these algorithms since we believe that the time taken by each algorithm is dependent on several variables such as the machine specifications, implementation details, and data type. Instead, we present these results as a qualitative analysis to show the relative performance of each algorithm as some parameter is varied.

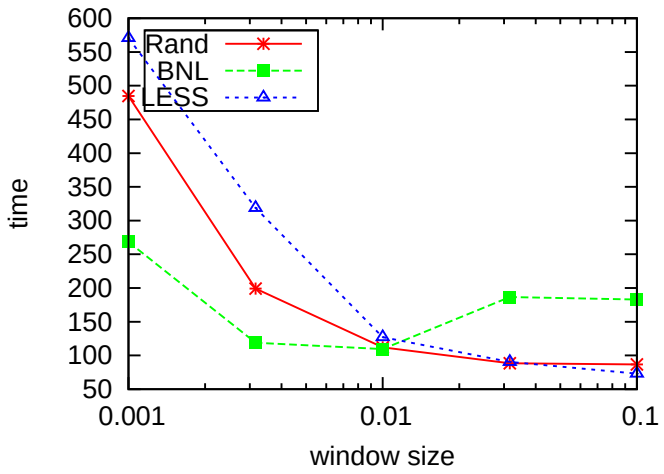
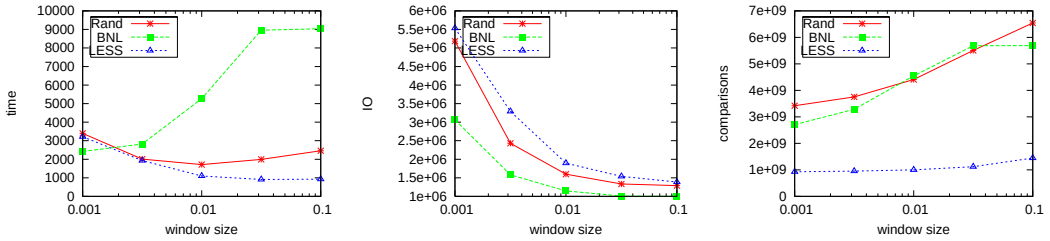


Figure 20: Varying w as a fraction of n for House

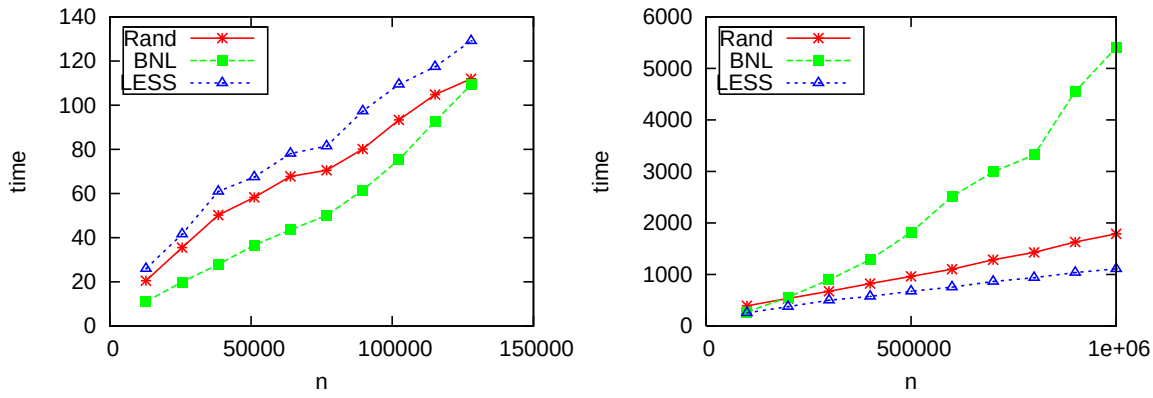


(a) Time (b) I/O's (c) Comparisons

Figure 21: Varying w as a fraction of n for Anti-correlated

In Figures 20 and 21, we show the performance of the different algorithms when the window size w (i.e., the number of points that can fit in main memory) is varied between 0.1% and 10% of the number of points n in the datasets. We show the graphs of I/O's and comparisons only for the anti-correlated case. The trends are similar for the house data. Since the number of I/O's (Figure 21(b)) decreases rapidly and the number of comparisons (Figure 21(c)) increases very slowly, RAND and LESS benefit from the larger window size. However, beyond some point, there is only marginal benefit. The running time of BNL increases as the window is made larger beyond some point. This is because, as the window size increases, the number of comparisons increases and BNL has book-keeping costs associated with each comparisons.

In Figure 22, we varied the number of data points when the window size is fixed



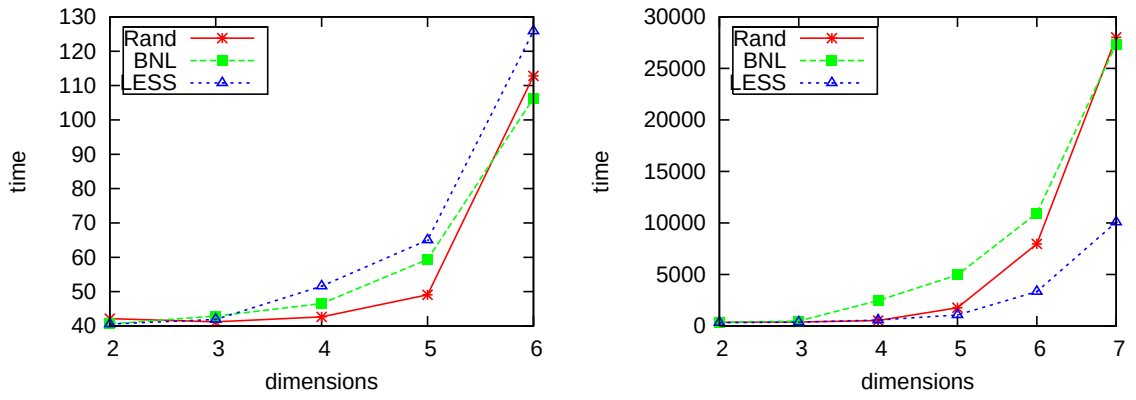
(a) House (b) Anti-correlated

Figure 22: Varying the number of points n

to 1%. We observe that the performance in all three metrics of all three algorithms is roughly linear in the number of points. We only show the plots for time for brevity. Whereas BNL is the best in the House dataset and LESS does well for the anti-correlated data, our algorithm performs comparably with both of them as n is increased.

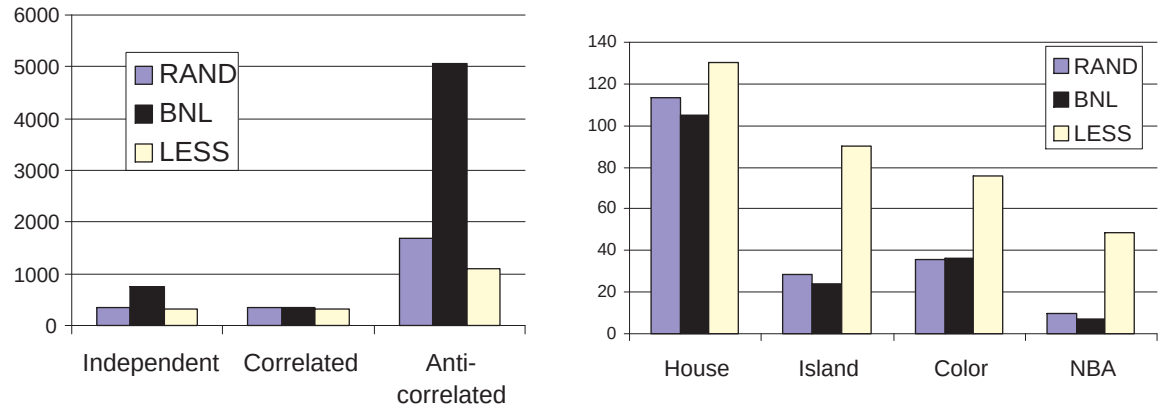
In Figure 23, we studied the variation of the performance in terms of dimensions. For House dataset, we pruned it down to the first two, three, four, five, and finally all six dimensions and compared the performance of the algorithms. All algorithms' performance in all metrics sharply increases, with different rates for each algorithm, when the number of dimensions goes beyond 5. This is because the number of skyline points increases rapidly as the number of dimensions increases. This is similar to results observed in [65] for the independent dataset. The fact that LESS becomes better than BNL and RAND as the dimensions are increased (equivalently the number of skyline points increased), but BNL and RAND beat LESS when the number of points are scaled can also be predicted from the comparisons table in Section 6.1.4.

From these results it is apparent that the performance of RAND is comparable to BNL and LESS. We observed similar performance on the other real and synthetic data sets.



(a) House (b) Anti-correlated

Figure 23: Varying the number of dimensions d



(a) Synthetic (b) Real

Figure 24: Running time on various datasets

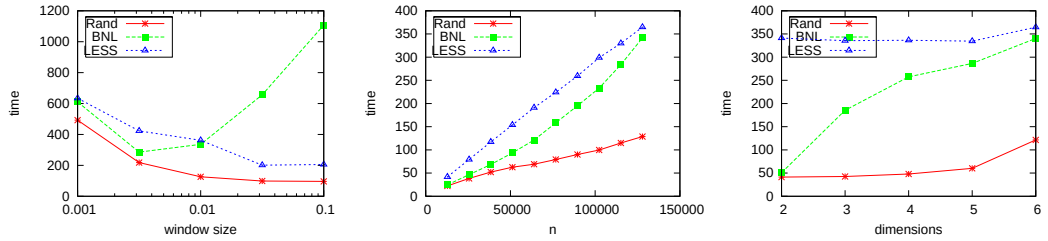
The running time of all algorithms in all datasets is summarized in Figure 24. Roughly speaking, LESS performs very well in the synthetic datasets (Figure 24 (a)) while BNL's performance can be sometimes very poor. However, BNL performs very well in all real datasets (Figure 24 (b)) while LESS' performance is always the worst and sometimes very poor. It could be concluded from these results that the performance of BNL and LESS relies heavily on the structure of input data. However, RAND is always close to the best algorithm. This could be partly because it has a good worst case guarantee.

6.4.2 Order Dependence

Next, we show how the performance of the other algorithms degrade considerably with a simple re-ordering of the data, whereas our algorithm's performance remains robust against these changes. We perform the following simple modifications of the data sets described above: in each case, we sort the data in decreasing order of entropy. Defined in [32], entropy is a scoring function that determines the likelihood with which a point is in the skyline. Additionally, we plot graphs where we sort the data in decreasing order of the entry in the first dimension.

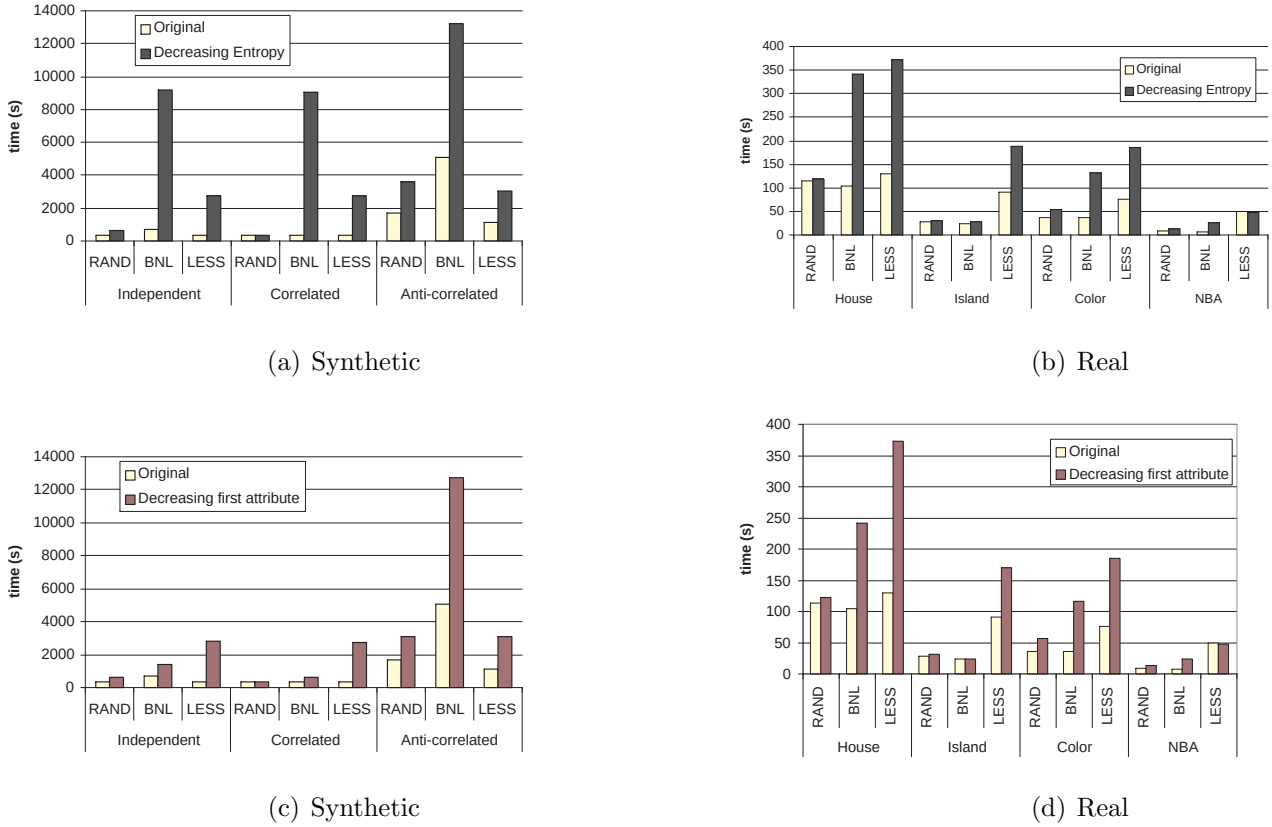
Figure 26 shows the comparative times for all three algorithms when the data is unsorted and sorted in decreasing entropy order. In almost all the cases, our algorithm shows little to no variance in performance. On the other hand, BNL and LESS both degrade in performance upto a factor of 2 or 3 times of their original running times for the real datasets and almost an order of magnitude difference for the larger synthetic datasets! This behavior can be explained as follows. By sorting in decreasing entropy order, we invalidate the effect of the entropy-based elimination filter of LESS. This makes LESS equivalent to SFS whose sorting phase incurs a lot of random I/O's and makes the algorithm much slower. In the case of BNL, the points which dominate fewer points are more likely to be found first in the stream and hence make their way in to the window. As a result, not many points are deleted from the stream in each pass. RAND, on the other hand, is still able to eliminate many points quickly as the skyline points in each phase are found starting from randomly sampled points.

In Figure 25, we vary w as a fraction of n, and vary d and n on the House data and test the performance of all three algorithms when sorted in decreasing entropy. The point of this experiment is to see whether the trends reflected in Figure 26 are an exception or whether this trend is expected for various settings (in terms of sizes and choice of parameters). As can be seen, in Figure 25 (b) and (c), as n or d is increased, the performance of BNL and LESS remain poor while RAND continues to



(a) Vary w (b) Vary n (c) Vary d

Figure 25: House, sorted by decreasing entropy. Varying w as a fraction of n, and varying d, n.



(a) Synthetic (b) Real

(c) Synthetic (d) Real

Figure 26: Variation in times for real and synthetic data sets (entropy sorted)

perform a few times better. Figure 25 (a) also shows the same trend but is a little confusing to interpret. The reason BNL climbs up rapidly after a point is because of what we had seen in Figure 20 due to the book-keeping cost increasing. However, the message here is that even if this factor is ignored, RAND is consistently better than both BNL and LESS, and therefore more resistant to perturbations in the input.

A similar variation is seen in the case of sorting in decreasing order by first dimension, see Figure 26 (c) and (d). While in the real data the variation of all three algorithms is similar to the entropy sorted case, for synthetic data, the variation is slightly less. Regardless, RAND remains the most robust to such minor alteration while the performance of both BNL and LESS degrade by a factor between about 2 and 10, depending on the case. We therefore believe that RAND is a good choice when real time performance requirements are stringent. It is hard to imagine a situation where a database sorts all records based on their entropy, however, many databases indeed store the records sorted by an attribute value. While these experiments only show trends on sorting by simple rules, it is conceivable that by changing the data, the performance fluctuates even more.

6.5 Distributed algorithm

We extend our algorithm to the distributed setting in the $\mathcal{CONGEST}$ model (cf. Chapter 1), where the database is *horizontally partitioned*, i.e., the set of points is partitioned across several machines. We note that we assume that at least one point can be sent through an edge in each round. Our goal is to minimize the number of rounds.

We start by making Eliminate-Points algorithm (cf. 6.1.1) suitable for the distributed setting. The main difficulty is that Pass 2 relies on the input order. We modify Pass 2 slightly to make it *order-independent*, as follows. (Note, however, that this version loses its generality. For example, it cannot solve the problem on posets

discussed in Section 6.3.2.)

Pass 2: For each p'_j, pick p_i in the stream that dominates p'_j and is *lexicographically* maximum (break tie by picking the first such points in the stream).

We call this algorithm Lex-Eliminate-Points. We claim that this algorithm also has the same bound guarantee as the Eliminate-Points algorithm (as in Lemma 6.4). Since the proof is essentially the same (with the definitions of "big sets" slightly modified), we omit the details.

Remark: The lexicographic order can be replaced by any *total* ordering that preserves domination. For example, if the input data is a set of vectors, we can pick p_i that dominates p'_j and has *maximum summation of values over all coordinates*, or *maximum entropy*, instead.

Now, consider the following distributed version of the Lex-Eliminate-Points algorithm (shown in Algorithm 6.5). As in the case of streaming algorithms, this algorithm has three passes. In the first pass, it samples $x = \lceil m \ln(mn \log n) \rceil$ points from the network. To do this, we use the idea of *reservoir sampling* that we used in Algorithm Sample-Coupon (cf. Algorithm 2.1 in Chapter 2). That is, we construct a Breadth-First-Search tree (of depth D, the diameter of the network) and sample each point based on the number of points in each subtree. All sampled points will be aggregated at the root node, denoted by $\mathcal{C}$. We do this using Algorithm Sample-Points (cf. Algorithm 6.5). In the second pass, all sampled points are broadcasted to all nodes in the network. For each sampled point, each node returns the *lexicographically* maximum point in its BFS subtree to its parent. These points are then aggregated at $\mathcal{C}$. Finally, $\mathcal{C}$ broadcast these points and each node deletes the points that are dominated by these points.

To analyze the running time of Algorithm Distributed-Lex-Eliminate-Points,

Algorithm 6.4 DISTRIBUTED-LEX-ELIMINATE-POINTS(m)

1: Let k machines be denoted by $\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_k$. Let n'_i be the current number of points in $\mathcal{M}_i$, and denote points by $p^i_1, p^i_2, ..., p^i_{n'_i}$ (in order). Let $x = \lceil m \ln(mn \log n) \rceil$.
2: Let $\mathcal{C}$ be any machine. Construct T, a Breadth-First-Search tree rooted at $\mathcal{C}$ (cf. Algorithm 6.5).
3: **Pass 1:** $\mathcal{C}$ samples x random points by calling Algorithm SAMPLE-POINTS($\mathcal{C}, T$) for x times.
4: **Pass 2:** $\mathcal{C}$ broadcasts $p_1, p_2, ..., p_x$ to all machines. For each $j = 1, 2, ..., x$, each machine M_i returns the *lexicographically* maximum point in the subtree of T rooted at it that dominates p_j. ($\mathcal{M}_i$ does this by waiting for points from all children and then return the lexicographically point among points received from its children and its own points.) Therefore, for each j, $\mathcal{C}$ has the point that is lexicographically maximum. Let the set $S' = \{p'_1, p'_2, ..., p'_x\}$ denote points kept by $\mathcal{C}$.
5: **Pass 3:** $\mathcal{C}$ broadcasts S' . Each machine deletes from its stream all points in or dominated by S'.
6: return S'

observe that Algorithm SAMPLE-POINTS finishes in $O(D)$ rounds since each node will send one point to its parent after it receives points from its children (i.e., the dilation is D and the congestion is one). Thus, Pass 1 can be done in $O(D + x)$ time. Similarly, for each point p_i, each machine $\mathcal{M}_j$ will return only one point that dominates it (i.e., the point that is lexicographically maximum). Thus, Pass 2 takes $O(D + x)$ time. Finally, broadcasting x points in Pass 3 also takes $O(D + x)$ times. It can be observed that the DISTRIBUTED-LEX-ELIMINATE-POINTS algorithm gives the same result as the LEX-ELIMINATE-POINTS algorithm on the stream defined above. Thus, we get the following theorem.

Lemma 6.16. *The* DISTRIBUTED-LEX-ELIMINATE-POINTS *algorithm uses* $O(x+D)$ *rounds of communication, where* $x = \lceil m \ln(mn \log n) \rceil$. *Moreover, after the algorithm finishes, at least half of the input data will be eliminated with probability at least* $1 - 1/(n \log n)$.

Using the above lemma, one can modify algorithms STREAMING-RAND to get a distributed algorithm as in the following theorem (similar to Theorem 6.6).

Algorithm 6.5 SAMPLE-POINTS($\mathcal{C}$, T)

Input: Starting node (machine) $\mathcal{C}$ and a breath-first-search tree T rooted at $\mathcal{C}$.
Output: A point sampled from among the points held by nodes in $T(v)$, the subtree of T rooted at v.

1: We divide T naturally into levels 0 through D (where nodes in level D are leaf nodes and the root node $\mathcal{C}$ is in level 0).
2: Every node u that holds some points picks one point uniformly at random. Let P_0 denote such point and let x_0 denote the number of points u has.
3: **for** $i = D$ down to 0 **do**
4: Every node u in level i that either receives points from children or possesses points itself do the following.
5: Let u have q points (including its own points). Denote these points by $P_0, P_1, P_2, \ldots, P_q$ and let their counts be $x_0, x_1, x_2, \ldots, x_q$. Node u samples one of P_0 through P_q, with probabilities proportional to the respective counts. That is, for any $1 \leq j \leq q$, P_j is sampled with probability $\frac{x_j}{x_0+x_1+\ldots+x_q}$.
6: The sampled point is sent to the parent node (unless already at root) along with a count of $x_0 + x_1 + \ldots + x_q$ (the count represents the number of points from which this coupon has been sampled).
7: **end for**

Theorem 6.17. *There is a distributed skyline algorithm in the $\mathcal{CONGEST}$ model that computes the skyline in $O(x \log n) = \tilde{O}(m)$ rounds with probability at least $1-1/n$.*

We note that this algorithm is almost optimal since delivering all skyline points to the central machine already takes $O(m)$ rounds. (One can also adapt the proof of Theorem 6.9 to show a lower bound that is slightly tighter.)

6.6 Conclusions

We present the first randomized streaming algorithm, RAND, for skyline computation. RAND has provable worst case guarantees on the number of sequential and randomized I/O's and number of pairwise comparisons. We show that it is optimal to within a logarithmic factor in terms of the space and passes used. We present a distributed version of RAND and a deterministic version for the 2-dimensional case. Finally, we experimentally evaluate the performance of RAND to LESS and BNL and show that it is comparable in the average case. Further, RAND is robust to minor

variations in the input while the performance of LESS and BNL deteriorate significantly. We believe that for applications where running time guarantees are desirable on skyline queries, RAND is the best choice.

We present the most simple version of RAND, however, variations of the algorithm may be more suitable for specific scenarios such as low-cardinality domains or specific input distributions etc. Further, a good question is whether a modification of RAND can be made to handle real-time user preferences. Finally, handling dynamic streams (additions and deletions of points) is an interesting setting that RAND does not yet adapt to.

Related publications. The preliminary version of this chapter appeared as a joint work with Atish Das Sarma, Ashwin Lall, and Jun Xu [42].

CHAPTER VII

GRAPH ALGORITHMS ON BEST-ORDER STREAMS

In this chapter, we partially answer whether there are efficient streaming algorithms when the input is in the best order possible. We give a negative answer to this question for the deterministic case and show evidence of a positive answer for the randomized case.

For the negative answer, we show that the space requirement is too large even for the simple problem of checking if a given graph has a perfect matching deterministically. In contrast, this problem, as well as the connectivity problem, can be solved efficiently by randomized algorithms. We show similar results for other graph properties.

Organization: This chapter is organized as follows. In Section 7.1 we describe our best-order streaming model formally and also define some of the other communication complexity models that are well-studied. The problem of checking for distinctness in a stream of elements is discussed in Section 7.2. This is a building block for most of our algorithms. The following section, Section 7.3, talks about how perfect matchings can be checked in our model. We discuss the problem of stream checking graph connectivity in Section 7.4. Our techniques can be extended to a wide class of graph problems such as checking for regular bipartiteness, non-bipartiteness, Hamiltonian cycles etc. We describe the key ideas for these problems in Section 7.5. Finally, we conclude in Section 7.6 by stating some insights drawn from this chapter, mention open problems and describe possible future directions.

7.1 Models

In this section we explain our main model and other related models that will be useful in the subsequent sections.

7.1.1 Best-Order Streaming Model

Recall the following classical streaming model which will be called the *worst-order* stream in this chapter, to contrast with the proposed best-order stream. In this model, an input is in some order $e_1, e_2, ..., e_m$, where m is the size of the input. The input $e_1, e_2, \ldots, e_m$ could be numbers, edges, or any other items. In this chapter, we are interested in the case where they are edges. We will assume this implicitly throughout. Moreover, we assume that the input element is indivisible (e.g., vertices in e_i must appear consecutively). In the case of graph problems considered in this chapter, we also assume that the number of vertices is known to the algorithm before reading the stream. (We note that the algorithms presented in this chapter also work even when we assume that the number of vertices are known only approximately.)

Consider any function f that maps the input stream to $\{0, 1\}$. The goal of the typical one-pass streaming model is to develop an algorithm that uses small space to read the input in order $e_1, e_2, \ldots, e_m$ and calculate $f(e_1, e_2, \ldots, e_m)$.

In the *best-order streaming* model, we consider any function f that is *order-independent*. That is, for any permutation π,

$$f(e_1, e_2, \ldots, e_m) = f(e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(m)}) .$$

Note that many graph properties (including those considered in this chapter) satisfy the above property. Our main question is how much space a one-pass streaming algorithm needs in order to compute f if the input is provided in the best order possible. Formally, for any function $s(m)$ and any function f, we say that a language L determined by f is in the Stream-Proof($s(m)$) class if there exists a streaming algorithm $\mathcal{A}$ with space $s(m)$ such that

- if $f(e_1, e_2, ..., e_m) = 1$ then there exists a permutation π such that $\mathcal{A}(e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(m)})$ answers 1;

- otherwise, $\mathcal{A}(e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(m)})$ answers 0 for *every* permutation π.

The other way to interpret this model is to consider the situation where there are two players in the setting, the *prover* and the *verifier*. The job of the prover is to provide the stream in some order so that the verifier can compute f using the smallest amount of memory possible. We assume that the prover has unlimited power but restrict the verifier to read the input in a streaming manner (with a limited memory).

The model above can be generalized to the following models.

- STREAM(p, s): A class of problems that, when presented in the best-order, can be checked by a deterministic streaming algorithm $\mathcal{A}$ using p passes and $O(s)$ space.

- RSTREAM(p, s): A class of problems that, when presented in the best-order, can be checked by a randomized streaming algorithm $\mathcal{A}$ using p passes and $O(s)$ space. The output is correct with probability at least $2/3$.

It is important to point out that when the input is presented in a specified order, we still need to check that the oracle is not *cheating*. That is, we indeed need a way to verify that we receive the input based on the rule we asked for. This often turns out to be the difficult step.

To contrast this model with the well-studied communication complexity models, we first define a new communication complexity model called *magic-partition* communication complexity. We later show a relationship between this model and the best-order streaming model.

7.1.2 Magic-Partition Communication Complexity

Recall the following standard 2-player communication complexity which we call *worst-partition* communication complexity. In this model, an input S, which is the set of elements, is partitioned into two sets X and Y, which are given to Alice and Bob, respectively. Alice and Bob want to together compute $f(S)$, for some order-independent function f. In the *worst-partition* case, we consider the case when the input is partitioned in an adversarial way, i.e., we partition the input into X and Y in such a way that Alice and Bob have to communicate as many bits as possible.

For the *magic-partition* communication complexity, we instead consider the case when the input is partitioned in the *best* way possible. Formally, the magic-partition communication complexity consists of three players, the oracle, and Alice and Bob. An algorithm on this model consists of a function $\mathcal{O}$ (owned by the oracle) that partitions the input set $S = \{e_1, e_2, \ldots, e_m\}$ to two sets $X = \{e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(\lfloor m/2 \rfloor)}\}$ and $Y = \{e_{\pi(\lfloor m/2 \rfloor+1)}, e_{\pi(\lfloor m/2 \rfloor+2)}, \ldots, e_{\pi(m)}\}$ for some permutation π and a protocol $\mathcal{P}$ used to communicate between Alice and Bob. We say that an algorithm consisting of $\mathcal{O}$ and $\mathcal{P}$ has communication complexity $c(m)$, for some function c, if

- for an input S such that $f(S) = 1$, the protocol $\mathcal{P}$ uses $c(m)$ bits of communication and outputs 1 when it is run on the sets X and Y partitioned according to $\mathcal{O}$, and
- for an input S such that $f(S) = 0$, the protocol $\mathcal{P}$ uses $c(m)$ bits of communication and outputs 0 when it is run on *any* sets X and Y coming from any partition.

One way to think of this protocol is to imagine that there is an oracle who looks at the input and then decides how to divide the data between Alice and Bob so that they can compute f using the smallest number of communicated bits and Alice and

Bob have to also check if the oracle is lying. We restrict that the input data must be divided equally between Alice and Bob.

Example 7.1. Suppose that the input is a graph G. Alice and Bob might decide that the graph be broken down in a topological order, i.e., they traverse the vertices in topological order and order the edges by the time they first visit vertices incident to them. It is important to note the distinction that Alice and Bob actually have not seen the input; but they specify a *rule* by which to partition the input, when actually presented.

Note that this type of communication complexity should not be confused with the best-partition communication complexity (defined in the next section).

The magic-partition communication complexity will be the main tool to prove the lower bounds of the best-order streaming model. The following lemma is the key to prove our lower bound results.

Lemma 7.2. *For any function f, if the deterministic magic-partition communication complexity of f is at least s, for some s, then for any p and t such that $(2p-1)t < s$, $f \notin \textsc{Stream}(p,t)$.*

Proof. Suppose that the lemma is not true; i.e., f has a magic-partition communication complexity at least s, for some s, but there is a best-order streaming algorithm $\mathcal{A}$ that computes f using p passes and t space such that $(2p-1)t < s$. Consider any input $e_1, e_2, ..., e_n$. Let π be a permutation such that $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(n)}$ is the best ordering of the input for $\mathcal{A}$. Then, define the partition of the magic-partition communication complexity by allocating $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$ to Alice and the rest to Bob.

Alice and Bob can simulate $\mathcal{A}$ as follows. First, Alice simulates $\mathcal{A}$ on $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$. Then, she sends the data on her memory to Bob. Then, Bob continues simulating $\mathcal{A}$ using the data given by Alice (as if he simulates $\mathcal{A}$ on $e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(\lfloor n/2 \rfloor)}$ by

himself). He then sends the data back to Alice and the simulation of the second pass of $\mathcal{A}$ begins. Observe that this simulations needs $2p-1$ rounds of communication and each round requires at most t bits. Therefore, Alice and Bob can compute f using $(2p-1)t < s$ bits, contradicting the original assumption. □

Similarly, if the randomized magic-partition communication complexity of f is at least s, for some s, then for any p and t such that $(2p-1)t < s$, $f \notin \text{RSTREAM}(p,t)$. Note also that the converse of the above lemma clearly does not hold.

7.1.3 Related models

We now describe some previously studied communication complexity models that resemble ours.

7.1.3.1 Best-Partition Communication Complexity

The best-partition communication complexity model was introduced by Papadimitriou and Sipser [124] and heavily used for proving the lower bounds for many applications including VLSI (see [100, 33, 95] and references therein). (In fact, many early communication complexity results are in this model.)

In this model, Alice and Bob can pick how to divide the data into two parts of roughly equal size among them *before* they see the input. This means that they can decide that if an element e appears in the stream, who will get this element. After this decision, the adversary, knowing this partitioning rule, gives an input that makes them communicate the most.

We note the following distinction between this model and the magic-partition model. In this model the players have to pick how data will be divided before they see the input data. For example, if the data is the graph of n vertices then, for any edge (i,j), Alice and Bob have to decide who will get this edge if (i,j) is actually in the input data. However, in the magic-partition model, Alice and Bob can make a more complicated partitioning rule such as giving $(1,2)$ to Alice *if* the graph is

connected. (In other words, in the magic-partition model, Alice and Bob have an oracle that helps them decide how to divide an input *after* he sees it).

Similar to the magic-partition communication complexity, this model makes many problems easier to solve than the traditional worst-partition model where the worst partitioning is assumed. However, the magic-partition model adds more power to the algorithms. In fact, the best-partition makes some problems strictly easier than the worst-partition model and the magic-partition model makes some problems strictly easier than the best-partition model, as shown in the following two examples.

Example 7.3. Consider the set disjointness problem. In this problem, two n-bit vectors x and y that are characteristic vectors of two sets X and Y are given. Alice and Bob have to determine if $X \cap Y = \emptyset$. In other words, they want to know if there is a position i such that the i-th bits of x and y are both one.

In the randomized worst case communication complexity, it has been proved that Alice has to send roughly n bits to Bob when x is given to Alice and y is given to Bob. However, for the best-partition case, they can divide the input in the following way: Alice receives the first $n/2$ bits of x and y and Bob receives the rest. This way, each of them can check the disjointness separately and Alice only has to send one bit to Bob (to indicate whether her strings are disjoint or not). Therefore, this problem in the best-partition model is strictly easier than in the worst-partition model.

Example 7.4. Consider the connectivity problem. Hajnal et al. [73] show that the best-partition communication complexity of connectivity is $\Theta(n \log n)$. In contrast, we show that $O((\log n)^2)$ is possible in our model in this chapter. Therefore, this problem in the magic-partition model is strictly easier than in the best-partition model.

7.1.3.2 Nondeterministic Communication Complexity

In this model, Alice and Bob receive x and y respectively. An oracle, who sees x and y, wants to convince them that "$f(x, y) = 1$". He does so by giving them a proof. Alice and Bob should be able to verify the proof with a small amount of communication. This model is different from the magic-partition model in that additional information (the proof) is provided by the oracle.

Example 7.5. Let x and y be n-bit strings. Consider the function $f(x, y)$ which is 1 if and only if $x \neq y$. If a proof is allowed, it can simply be the number i where $x_i \neq y_i$. Then, Alice and Bob can check the proof by exchanging one bit (x_i and y_i). If $x = y$ then there is no proof and Alice and Bob can always detect the fake proof.

7.2 Detecting a Duplicate and Checking for Distinctness

In this section, we consider the following problem which is denoted by DISTINCT. Given a stream of n numbers $a_1, a_2, ..., a_n$ where $a_i \in \{1, 2, ..., n\}$, we want to check if every number appears exactly once (i.e., no duplicate). We are interested in solving this problem in the worst-order streaming model. This problem (in the worst-order model) appears to be a crucial component in solving all the problems we consider in the best-order streaming model and we believe that it will be useful in every problem.

Our goal in this section is to find a one-pass worst-order streaming algorithm for this problem. The algorithm for this problem will be an important ingredient of all algorithms we consider in this chapter. In this section, we show that

1. any deterministic algorithm for this problem needs $\Omega(n)$ space, and
2. there is a randomized algorithm that solves this problem in $O(\log n)$ space with an error probability at most $\frac{1}{n}$.

7.2.1 Space lower bound of the deterministic algorithms

Since checking for distinctness is equivalent to checking if there is a duplicate, a natural problem to consider as a lower bound is the *set disjointness problem*. We define a variation of this problem called *full set disjointness problem*, denoted by F-DISJ.

In this problem, a set $X \subseteq [n]$ is given to Alice and a set $Y \subseteq [n]$ is given to Bob where $[n] = \{1, 2, 3, ..., n\}$ and $|X| + |Y| = n$. Alice and Bob want to together check whether $X \cap Y = \emptyset$.

Note that this problem is different from the well-known set disjointness problem in that we require $|X| + |Y| = n$. Although the two problems are very similar, they are different in that the set disjointness problem has an $\Omega(n)$ lower bound for the randomized protocol in the worst-partition communication complexity model while the F-DISJ has a $O(\log n)$-communication randomized protocol (shown in the next section). We also note that the lower bound of another related problem called k-disjointness problem (see, e.g., [95, example 2.12] and [74]) does not imply the lower bound of F-DISJ shown here.

Now we show that F-DISJ is hard in the deterministic case. The proof is essentially the same as the proof of the lower bound of the set disjointness problem.

Theorem 7.6. *The communication complexity of* F-DISJ *is* $\Omega(n)$.

Proof. We use a standard technique called the fooling set technique. A fooling set is a set $F\{(A_1, B_1), (A_2, B_2), \ldots, (A_k, B_k)\}$ of size k such that $f(A_i, B_i) = 1$ for all i and $f(A_i, B_j) = 0$ for all $i \neq j$. Once this is shown, it will follow that the deterministic communication complexity is $\Omega(\log(|F|))$. (See the proof in, e.g., [95]).

Now, consider the fooling set $F = \{(A, N \setminus A) \ : \ \forall A \subseteq N\}$. It is easy to check that the property above holds. Since $|F| = 2^n$, the number of bits needed to sent between Alice and Bob is at least $\log(|F|) = \Omega(n)$. □

We note that the theorem also follows from the lower bound of the variation of EQUALITY (checking whether $X = Y$) where we let $Y = [n]$. The theorem implies the space lower bound of DISTINCT.

Corollary 7.7. *Any deterministic worst-order streaming algorithm for* DISTINCT *needs* $\Omega(n)$ *space.*

This lower bound is for the worst-order input. The reason we mention this here is because this seems to be an inherent difficulty in the algorithms in the best-order streaming model. As shown later, all algorithms developed in this chapter need to solve DISTINCT as a subroutine. In fact, for all these algorithms, solving DISTINCT is the only part that needs the randomness.

7.2.2 Randomized algorithm

In this subsection we present a randomized one-pass worst-order streaming algorithm that solves DISTINCT using $O(\log n)$ space. This algorithm is based on the *Fingerprinting Sets* technique introduced by Lipton [104, 105]. Roughly speaking, given a multi-set $\{x_1, x_2, ..., x_k\}$, its *fingerprint* is defined to be

$$\Pi_{i=1}^{k}(x_i + r) \mod p$$

where p is a random prime and $r \in \{0, 1, ..., p-1\}$. We use the following property of the fingerprints.

Theorem 7.8. *[105] Let* $\{x_1, x_2, ..., x_k\}$ *and* $\{y_1, y_2, ..., y_l\}$ *be two multi-sets. If the two sets are equal then their fingerprints are always the same. Moreover, if they are unequal, the probability that they get the same fingerprints is at most*

$$O(\frac{\log b + \log m}{bm} + \frac{1}{b^2 m})$$

where all numbers are b-bit numbers and $m = \max(k, l)$ *provided that the prime* p *is selected randomly from interval*

$$[(bm)^2, 2(bm)^2].$$

Now, to check if $a_1, a_2, ..., a_n$ are all distinct, we simply check if the fingerprints of $\{a_1, a_2, ..., a_n\}$ and $\{1, 2, ..., n\}$ are the same. Here, $b = \log n$ and $m = n$. Therefore, the error probability is at most $1/n$.

Remark We note that the fingerprinting sets technique can also be used in our motivating application of cloud computing above. That is, when the cloud sends back a graph as a proof, we have to check whether this "proof" graph is the same as the input graph we sent. This can be done by checking if the fingerprints of both graphs are the same. This enables us to concentrate on checking the stream without worrying about this issue in the rest of this chapter.

We also note that the recent result by Gopalan et al. [68] can be modified to solve DISTINCT as well. Finally, note that we need to know n, or its upper bound, before we run the algorithm.

7.3 Perfect Matching

We exhibit the ideas of developing algorithms and lower bounds in the best-order streaming model through the perfect matching problem.

Problem Let G be an input graph of n vertices where the vertices are labeled $1, 2, \ldots, n$. Given the edges of G in a streaming manner $e_1, e_2, ..., e_m$, we want to compute $f(e_1, ..., e_m)$ which is 1 if and only if G has a perfect matching. Let n be the number of vertices.

7.3.1 Upper Bound

Theorem 7.9. *The problem of determining if there exists a perfect matching can be solved by a randomized best-order streaming algorithm using $O(\log n)$ space with a success probability at least $1 - 1/n$.*

Proof. Consider the following algorithm.

Algorithm The prover sends $n/2$ edges of a perfect matching to the verifier first and then send the rest of the edges. The verifier then check the followings.

1. Check if the first $n/2$ edges form a perfect matching. This can be done by checking whether the fingerprint of the set $\bigcup_{i=1}^{n/2} e_i$ (where $e_1, e_2, \ldots, e_{n/2}$ are the first $n/2$ edges in the stream) is equal to the fingerprint of the set $\{1, 2, \ldots, n\}$.

2. Check if there are at most n vertices. This is done by checking that the maximum vertex label is at most n.

Finally, the verifier outputs 1 if the input passes all the above tests.

The correctness of this algorithm is quite straightforward, as follows. First, if the edges $e_1, e_2, \ldots e_{n/2}$ form a perfect matching then $e_1, e_2, \ldots e_{n/2}$ have no vertex in common and, therefore, $\bigcup_{i=1}^{n/2} e_i = \{1, 2, \ldots, n\}$. This means that the fingerprints of $\bigcup_{i=1}^{n/2} e_i$ and $\{1, 2, \ldots, n\}$ are always the same. Thus, the first condition holds. The second condition can be also easily checked. Therefore, the algorithm will output 1 in this case.

For the case that the edges $e_1, e_2, \ldots e_{n/2}$ do not form a perfect matching, observe that $\bigcup_{i=1}^{n/2} \neq \{1, 2, \ldots, n\}$ and therefore the fingerprints of the two sets will be different with probability at least $1-1/n$. Consequently, the algorithm will successfully output 0 with probability at least $1 - 1/n$. □

7.3.2 Lower Bound

We show that the deterministic best-order streaming algorithms for the perfect matching problem have $\Omega(n)$ lower bound *if the input is ordered in an explicit way*; i.e., each edge cannot be split. This means that an edge is either represented in the form (a, b) or (b, a). The proof follows from a reduction from the magic-partition communication complexity (cf. Section 7.1) of the same problem by using Lemma 7.2.

Theorem 7.10. *If the input can be reordered only in an explicit way then any deterministic algorithm solving the perfect matching problem needs $\Omega(n)$ space, where n is the number of vertices.*

Proof. Let n be any even integer divisible by four. We show that the above theorem is true even when the input always contains exactly $n/2$ edges. In this case, checking whether these $n/2$ edges form a perfect matching is equivalent to checking whether every vertex appears as an end vertex of exactly one edge. We note that the input is allowed to contain multiple edges. (However, such inputs clearly do not form perfect matchings over n vertices.)

We now show that the magic-partition communication complexity of the perfect matching problem is $\Omega(n)$. Once this is done, the theorem follows immediately by Lemma 7.2.

Consider any magic-partition communication complexity protocol which consists of a partition function $\mathcal{O}$ owned by an oracle and a communication protocol $\mathcal{P}$ between Alice and Bob. That is, a function $\mathcal{O}$ partitions the input into two sets of edges, A and B where $|A| = n/4$ and $|B| = n/4$. Then, A and B are sent to Alice and Bob, respectively. Alice and Bob, upon receiving A and B, communicate to each other using a protocol $\mathcal{P}$ and one of them outputs whether the input edges form a perfect matching or not ("YES" or "NO"). The main goal is to show that for any partition function $\mathcal{O}$, there is some input that forces $\mathcal{P}$ to incur $\Omega(n)$ bits of communication. Recall that $\mathcal{P}$ has to deal with the following cases: 1) If the input is a perfect matching, $\mathcal{P}$ has to output YES when the input is partitioned according to $\mathcal{O}$. 2) Otherwise, $\mathcal{P}$ has to output NO for *any* partition of the input. We now show the communication complexity of $\mathcal{P}$.

First, let us consider the inputs that are perfect matchings. Let $g(n)$ denote the

number of distinct perfect matchings in the complete graph K_n. Observe that

$$g(n) = \frac{n!}{(n/2)!2^{n/2}}.$$

Denote these matchings by $M_1, M_2, \ldots, M_{g(n)}$. For any integer i, let A_i and B_i be the partition of M_i according to $\mathcal{O}$. We now partition $M_1, \ldots, M_{g(n)}$ into clusters in such a way that the matchings whose vertices are partitioned in the same way are in the same cluster. That is, any two inputs M_i and M_j are in the same cluster if and only if $\bigcup_{e \in M_i} e = \bigcup_{e \in M_j} e$.

We claim that there are at least $\binom{n/2}{n/4}$ clusters. To see this, observe that for any matching M_i, there are at most $g(n/2)^2$ matchings that vertices *could be* partitioned the same way as M_i. (I.e., if we define $V(A_i) = \{v \in V \ : \ \exists e \in A_i \ s.t. \ v \in e\}$ then for any i, $|\{j \ : \ V(A_i) = V(A_j)\}| \leq g(n/2)^2$.) This is because $n/2$ vertices on each side of the partition can make $g(n/2)$ different matchings. This implies that the size of each cluster is at most $g(n/2)^2$. Therefore, the number of matchings such that the vertices are divided differently is at least

$$\frac{g(n)}{g(n/2)^2} = \frac{n!}{(n/2)!2^{n/2}} \left(\frac{(n/4)!2^{n/4}}{(n/2)!} \right)^2 = \binom{n}{n/2} / \binom{n/2}{n/4} \geq \binom{n/2}{n/4}$$

where the last inequality follows from the fact that $\binom{n}{n/2}$ is the number of subsets of $\{1, 2, ..., n\}$ of size $n/2$ and $\binom{n/2}{n/4}^2$ is the number of parts of these subsets.

Let t be the number of clusters (so $t \geq \binom{n/2}{n/4}$) and let $M_{i_1}, M_{i_2}, \ldots, M_{i_t}$ be the inputs from different clusters and let $(A_{i_1}, B_{i_1}) \ldots, (A_{i_t}, B_{i_t})$ be the corresponding partitions according to $\mathcal{O}$. Observe that for any $t' \neq t''$, an input consisting of edges in $M_{i_{t'}}$ and $M_{i_{t''}}$ is not a perfect matching. Moreover, observe that for any t' and t'', any pair $(A_{i_{t'}}, B_{i_{t''}})$ could be an input to the protocol $\mathcal{P}$ (since the oracle can partition the input in anyway when the input is not a perfect matching). In other words, the communication complexity of $\mathcal{P}$ is the *worst case* (in term of communication bits) among all pairs $(A_{i_{t'}}, B_{i_{t''}})$.

For readers who are familiar with the standard fooling set argument, it follows almost immediately that the communication complexity of $\mathcal{P}$ is $\Omega(\log t) = \Omega(n)$ and the theorem is thus proved. For those who are not familiar with this argument, we offer the following alternative argument.

Let $t' = \lfloor \log t \rfloor$. (Note that $t' = \Omega(n)$.) Consider the problem $\textsc{eq}_{t'}$ where Alice and Bob each gets a t'-bit vector x and y, respectively. They have to output YES if $x = y$ and NO otherwise. It is well known (see, e.g., [95, Example 1.21]) that the deterministic worst-partition communication complexity of $\textsc{eq}_{t'}$ is at least $t' + 1 = \Omega(n)$.

Now we reduce $\textsc{eq}_{t'}$ to our problem using the following protocol $\mathcal{P}'$: Upon receiving x and y, Alice and Bob locally map x to A_{i_x} and y to B_{i_y}, respectively and then simulate $\mathcal{P}$. Since $x = y$ if and only if $A_{i_x} \cup B_{i_y}$ is a perfect matching, $\mathcal{P}'$ outputs YES if and only if $x = y$. Therefore, Alice and Bob can use the protocol $\mathcal{P}'$ to solve $\textsc{eq}_{t'}$. Since, the deterministic worst-partition communication complexity of $\textsc{eq}_{t'}$ is $\Omega(n)$, so is the communication complexity of $\mathcal{P}$. This shows that the deterministic magic-partition communication complexity of the matching problem is $\Omega(n)$. □

Note that the above lower bound is asymptotically tight since we can check if there is a perfect matching using $O(n)$ space in the best-order streams: The oracle simply puts edges in the perfect matching first in the stream. Then, the algorithm checks whether the first $n/2$ edges in the stream form a matching by checking whether all vertices appear (using an array of n bits).

Also note that the following argument might lead to a wrong conclusion that the magic-partition communication complexity of Distinct is also $\Omega(n)$: If Distinct can be done in $o(n)$ bits by a magic-partition protocol, then we can put it in the protocol in Theorem 7.9 to solve the perfect matching problem using $o(n)$ bits. This will contradict Theorem 7.10.

However, the problem of the above argument is that the protocol in Theorem 7.9

needs the *worst-partition* communication complexity of DISTINCT. In fact, DISTINCT can be easily solved in 1 bit using the following magic-partition communication complexity protocol: The oracle sends the first $n/2$ smallest numbers to Alice and sends the rest to Bob. Alice sends 1 to Bob if her numbers are $1, 2, ..., n/2$ and Bob outputs YES if he receives 1 from Alice and his numbers are $n/2 + 1, n/2 + 2..., n$.

7.4 Graph Connectivity

Graph connectivity is perhaps the most basic property that one would like to check. However, even graph connectivity does not admit space-efficient algorithms in the traditional worst-order streaming model as there is an $\Omega(n)$ lower bound for randomized algorithms. To contrast this, we show that allowing the algorithm the additional power of requesting the input in a specific order allows for a very efficient, $O((\log n)^2)$-space algorithm for testing connectivity.

Problem We consider a function where the input is a set of edges and $f(e_1, e_2, ..., e_m) = 1$ if and only if G is connected. As usual, let n be the number of vertices of G. As before, we assume that vertices are labeled $1, 2, 3, ..., n$.

7.4.1 Upper Bound

We will prove the following theorem.

Theorem 7.11. *Graph connectivity can be solved by a randomized algorithm using $O((\log n)^2)$ space in the best-order streaming model.*

Proof. We use the following lemma constructively.

Lemma 7.12. *For any graph G of $n - 1$ edges, where $n \geq 3$, G is connected if and only if there exists a vertex v and trees $T_1, T_2, ..., T_q$ such that for all i,*

- *$\bigcup_{i=1}^{q} V(T_i) = V(G)$ and for any $i \neq j$, $V(T_i) \cap V(T_j) = \{v\}$, and*
- *there exists a unique vertex $u_i \in V(T_i)$ such that $u_i v \in E(T_i)$, and*

- $|V(T_i)| \leq \lceil 2n/3 \rceil$ *for all* i.

Proof. To see this proof, notice that G is a spanning tree since it is connected and has exactly $n-1$ edges. Consider any vertex in the spanning tree, which on deleting, disconnects the graph in to two or more pieces such that each piece has at most $2n/3$ vertices. The existence of such a vertex can be proven by induction. The base case where $n = 3$ can be proved simply by picking a vertex in the middle of the path of length 3. Suppose such a vertex exists for all $n' < n$. Consider a tree on n vertices. Now, remove a leaf node, say z, and, by induction, such a vertex v exists on the tree on remaining n vertices. Now add z back and let C be the component (on deleting v) that contains z. If C has size at most $\lceil 2n/3 \rceil - 1$, the same v works on the larger tree. If C has size at least $\lceil 2n/3 \rceil$ then consider the unique vertex in this component that connects to v, say u. Observed that u serves as the new vertex for the lemma. This is because the complement of C together with u has size at most $n - (\lceil 2n/3 \rceil) + 1 \leq 2n/3$. Since this can be done, u can be chosen as a vertex for the lemma. Whenever a vertex in a tree is disconnected, the new components also form trees. Call these $T_1, T_2, \ldots T_q$. Notice that there can be at most one vertex adjacent to v in each component T_i, since G has no cycles. Call this vertex in T_i by u_i. Therefore each $u_i \in T_i$ is adjacent to u and each T_i has at most $\lceil 2n/3 \rceil$ nodes. □

It is sufficient to consider graphs of $n-1$ edges, as these $n-1$ edges that form a connected spanning component are sufficient to verify connectivity. Suppose that G is connected, i.e., G is a tree. Let v and $T_1, T_2, ..., T_q$ be as in the lemma. Define the order of G to be

$$Order(G) = vu_1, Order(T_1'), vu_2, Order(T_2'), ..., vu_q, Order(T_q')$$

where $T_i' = T_i \setminus \{vu_i\}$. Note that T_i' is a connected tree and so we present edges of T_i' recursively. The recursion step ends when the eventual subtree is a star, i.e., edges presented are $vu_1, vu_2, \ldots$. At this point, the verifier just checks that all consecutive

edges are adjacent to the same vertex and form a star. This depth of recursion can be checked directly.

Now, when edges are presented in this order, the checker can check if the graph is connected as follows. First, the checker reads vu_1. The checker remembers the vertex v, which takes $O(\log n)$ bits. Then the edges in T_1' are presented. He checks if T_1' is connected by running the algorithm recursively. Note that he stops checking T_1' once he sees vu_2. Notice that this step is consistent since the vertex v does not appear in any T_i'. Once an edge with a vertex v is received, the checker knows that the tree has been verified and the next tree is to be presented. So the checker repeats with vu_2 and T_2' and so on. Here again, v does not appear in T_2' but u_2 does. Therefore, the checker now again needs to check that T_2' is connected. Further, it is automatically checked that T_2' connects to v due to the edge vu_2. The checker proceeds in this manner checking the connectivity of each T_i' up to $i = q$. If each tree is connected (which is checked recursively), and all the edges vu_i appear separating the trees, then all the trees are connected to v. Therefore, the entire set of edges presented is connected. However, this does not guarantee that n distinct vertices, or n distinct edges have been received. Therefore, it only remains to be checked that n distinct edges have been presented.

He does so by applying the result in Section 7.2 once to each vertex v used as a root (as in above) and all leaf nodes of the tree. If all n distinct vertices have appeared at least once, and the set of first n edges form a connected component, then G is a connected graph. Also note that if G is not connected then such ordering cannot be made and the algorithm above will detect this fact.

The space needed is for vu_i and for checking T_i'. I.e., $space(|G|) = space(\max_i |T_i|) + O(\log n)$. That is, $space(n) \leq space(\lceil 2n/3 \rceil) + O(\log n)$. This gives the claimed space bound. □

7.4.2 Lower Bound

Recall that we say that the input is ordered in an explicit way if each edge is presented in the form (a, b) where a and b are the labels of its end vertices.

Theorem 7.13. *If the input can be ordered only in an explicit way, any deterministic algorithm solving the connectivity problem on the best-order stream needs* $\Omega(n \log n)$ *space, where* n *is the number of vertices.*

Proof. Let n be an odd number. We show that the theorem holds even when the input always consists of exactly $n - 1$ edges. (Therefore, the task is only to check whether the input edges form a spanning tree over n nodes.) We show this via the magic-partition communication complexity. Since the argument is essentially the same as that in the the proof of Theorem 7.10, we only give the essential parts here.

Assume that n is an odd number more than two. Let $g(n)$ be the number of spanning trees of the complete graph K_n. By Cayley's formula (see, e.g., [4]),

$$g(n) = n^{n-2}.$$

Let $T_1, T_2, \ldots, T_{g(n)}$ denote such trees. Consider any best-partition communication complexity protocol which consists of a partition function $\mathcal{O}$ owned by the oracle and a protocol $\mathcal{P}$ used by Alice and Bob. For $i = 1, 2, \ldots, g(n)$, let (A_i, B_i) be the partition of the input edges of T_i to Alice and Bob, respectively, according to $\mathcal{O}$.

Now, draw a graph H consisting of $g(n)$ vertices, $v_1, v_2, \ldots, v_{g(n)}$. Draw an edge between vertices v_i and v_j if $A_i \cup B_j$ or $A_j \cup B_i$ is a spanning tree.

We claim that each vertex in H has degree at most $2g((n + 1)/2)$. To see this, observe that for any set A of $(n-1)/2$ edges, there are at most $g((n+1)/2)$ sets B of $(n-1)/2$ edges such that $A \cup B$ is a spanning tree. This is because when we contract edges in A, there are $(n + 1)/2$ vertices left and these vertices must form a spanning tree on the contracted graph. This observation is also true when we look at the set B. The claim thus follows.

Now pick an independent set from H using the following algorithm: Pick any vertex, delete such vertex and its neighbors and then repeat. Observe that this algorithm gives an independent set of size at least $\frac{g(n)}{2g((n+1)/2)+1}$ since there are $g(n)$ vertices in H and each vertex has degree at most $2g((n+1)/2)$. Let t be the size of the independent set. Note that $t = \frac{g(n)}{2g((n+1)/2)+1} = n^{\Omega(n)}$. Let $v_{i_1}, v_{i_2}, \ldots, v_{i_t}$ be the vertices in the independent set.

Consider the trees $T_{i_1}, T_{i_2}, \ldots, T_{i_t}$ corresponding to the independent set picked by the above algorithm. Since there is no edge between any $v_{i_{t'}}$ and $v_{i_{t''}}$, $(A_{i_{t'}}, B_{i_{t''}})$ and $(A_{i_{t''}}, B_{i_{t'}})$ do not form a spanning tree. As argued in the proof of Theorem 7.10, the protocol $\mathcal{P}$ must be able to receive any pair of the form $(A_{i_{t'}}, B_{i_{t''}})$ and answer YES if and only if $t' = t''$. By the fooling set argument or the reduction from $\text{EQ}_{\log t}$, it follows that $\mathcal{P}$ needs $\Omega(\log t) = \Omega(n \log n)$ bits, as desired. □

We note that the lower bound above is asymptotically tight since we can solve connectivity problem using the following $O(n \log n)$-space deterministic algorithm: The oracle present edges in a spanning tree first in the stream. Then the algorithm reads and checks whether these edges form a spanning tree using $O(n \log n)$ space.

7.5 Further Results

The previous sections give us a flavor of the results that can be obtained in the best-order streaming model. We describe a few more and mention the intuition behind the protocol without going into details since the techniques are essentially the same.

7.5.1 Bipartite k-Regular graph

The problem is to check if the graph is bipartite k-regular. First, note that since this problem is the generalization of the perfect matching problem (cf. Section 7.3), the $\Omega(n)$ lower bound of the deterministic algorithms holds here. Now we show that this problem can be solved by a randomized algorithm with $O(\log n)$ space.

The point of the algorithm is that a k-regular bipartite graph can be decomposed into k disjoint perfect matchings. So the oracle can do this and present each of the perfect matchings one after the other. However, as it will be clear soon, the oracle has to send each edge in the form (a, b) where a is the "left" vertex and b is the "right" one. This forces the algorithm to find another way to find out the value of n (instead of looking for a "flip" edge as used by the perfect matching algorithm).

The algorithm can find out n in the following way: While reading the first perfect matching, it remembers the maximum vertex label it saw so far, denoted by n'. Once the number of edges it read so far equals $n'/2$ (for the current value of n'), it looks one more edge further. If this edge consists of vertices with labels at most n' then it concludes that this value of n' is the value of n. The correctness of this method can be seen by observing that if $n' < n$ and no vertex appears twice in the first $n'/2$ edges then the labels of vertices in the next edge must be both more than n'.

Now, we describe the last part of the algorithm. It has to verify the following.

1. Each set of $n/2$ edges form a perfect matching. This can be verified separately for each set of $n/2$ edges.

2. In each matching, it sees the same set of "left" vertices and "right" vertices. This can be done by computing the finger prints of the sets of left and right vertices.

Note that the reason that the oracle has to present the edges in the form of left and right vertices is to allow the algorithm to check the second condition.

7.5.2 Hamiltonian Cycle

The problem is to check whether the input graph has a Hamiltonian cycle. We claim that this problem is in RSTREAM$(1, \log n)$. The intuition is for the oracle to provide the Hamiltonian cycle first (everything else is ignored). The algorithm then checks if

the first n edges indeed form a cycle; this requires two main facts. First that every two consecutive edges share a vertex, and the n-th edge shares a specific vertex with the first. This fact can be easily checked. The second key step is to check that these edges indeed span all n vertices (and not go through same vertex more than once). This can be done by the fingerprinting technique.

We also claim that there is an $\Omega(n)$ lower bound for the deterministic algorithms if the edges can be ordered only in an explicit way. The proof is essentially similar to the proof of other lower bounds shown earlier and we only sketch it here. We consider the inputs that have exactly n edges. Let $g(n)$ be the number of the Hamiltonian cycles covering n vertices. Clearly $g(n) = (n-1)!$. Let $C_1, \ldots C_{g(n)}$ be these cycles and $(A_1, B_1), \ldots, (A_{g(n)}, B_{g(n)})$ be the corresponding partitions. Since after we see $n/2$ edges, there could be only $g(n/2)$ possible Hamiltonian cycles containing these edges, we can pick $\frac{g(n)}{2g(n/2)+1}$ that are "independent" in the same sense as in Theorem 7.13. The communication complexity is thus

$$\Omega\left(\log\left(\frac{g(n)}{2g(n/2)+1}\right)\right) = \Omega(n)\,.$$

7.5.3 Non-Bipartiteness

The problem is to check if the graph is not bipartite. This problem can be solved by a *deterministic* algorithm with $O(\log n)$ space by having an oracle present an odd length (not necessarily simple) cycle. Verifying that this is indeed a cycle and that it is of odd length can be done easily.

In contrast to the Non-bipartiteness problem, we do not have an algorithm for checking the *bipartiteness* of graphs. We conjecture that this problem has a super-logarithmic randomized lower bound. We note, however, that if we relax the model by allowing the input to be presented twice then there is an efficient randomized algorithm: Let $U = \{u_1, u_2, \ldots, u_{n'}\}$ and $V = \{v_1, v_2, \ldots, v_{n''}$ be the two partitions. In the first rounds, present edges incident to u_1 first then present edges incident to u_2,

and so on (i.e., edges are presented in the form (u_1, v_{i_1}), (u_1, v_{i_2}), ..., $(u_2, v_{i'_1})$, $(u_2, v_{i'_2})$, ..., $(u_{n'}, v_{i''_1})$, ...) Similarly, in the next round present edges incident to v_1, v_2, ..., $v_{n''}$, respectively, with vertices in V appearing first (i.e., in the form $(v_1, u_{i_1}), (v_1, u_{i_2}), \ldots$). We can use the fingerprinting technique to check if $u_1, \ldots, u_{n'}, v_1, \ldots, v_{n''}$ are all distinct.

7.6 Conclusions

This chapter describes a new model of stream checking that lies at the intersection of several extremely well-studied and foundational fields of computer science. Specifically, the model connects several settings related to proof checking, communication complexity, and streaming algorithms. The motivation of this work, however, arises from the recent growth in the data sizes and the advent of the powerful cloud computing architectures and services. The question we ask is, can the verification of certain properties (on any input) be accompanied with a streaming proof of the fact? The checker should be able to verify that the prover is not cheating. We show that if the checker (or the algorithm in the best-order streaming setting) is given the power of choosing a specific rule for the prover to send the input, then many problems can be solved much more efficiently in this model than in the previous models.

While non-obvious, our algorithms and proofs are fairly simple. However, the nice aspect is that it uses several interesting techniques from many areas such as fingerprinting and covert channels. Fingerprinting is used in a crucial way to randomly test for the distinctness of a set of elements presented as a stream. The protocol between the prover and the checker also allows for a covert communication (which gives the covert channels a positive spin as opposed to the previous studies in security and cryptography). While the prover is only allowed to send the re-ordered input, the prover is able to encode some extra bits of information with the special ordering requested by the checker. The difficulty in most of our proof techniques is in how the

checker or algorithm verifies that the prover or oracle is sending the input order as requested.

We have given randomized $O(\text{polylog}\, n)$-space algorithms for problems that previously, in the streaming model, had no sub-linear space algorithms. We note that in all protocols presented in this chapter, the prover can construct the best-order proofs in polynomial time. There are still a lot of problems in graph theory that remain to be investigated. A nice direction is to consider testing for graph minors, which could in turn yield efficient methods for testing planarity and other properties that exclude specific minors. It is also interesting to see whether all graph problems in the complexity class P can be solved in our model with $O(\text{polylog}\, n)$ space. Such a result would be a huge improvement over the result in [105] (which needs a proof of size near-linear in the number of steps for the computation) in terms of the proof size for graph problems. (One good starting point are problems of checking bipartiteness, non-connectivity, and non-existence of perfect matching.) Moreover, it is interesting to see whether additional passes would be of much help. Additionally, is the "flipping trick" necessary? That is, if we present each edge as a set $\{u, v\}$ instead of an ordered pair (u, v), do efficient protocols for the problems presented here still exist?

Apart from the study of our specific model, we believe that the results and ideas presented in this chapter could lead to improved algorithms in the previously studied settings as well as yield new insights to the complexity of the problems.

Related publications. The preliminary version of this chapter appeared as a joint result with Atish Das Sarma, and Richard J. Lipton [43].

REFERENCES

[1] "Amazon elastic compute cloud (amazon ec2)."

[2] Adamic, L. A., Lukose, R. M., Puniyani, A. R., and Huberman, B. A., "Search in power-law networks," *Physical Review*, vol. 64, 2001.

[3] Aggarwal, G., Datar, M., Rajagopalan, S., and Ruhl, M., "On the streaming model augmented with a sorting primitive," in *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, (Washington, DC, USA), pp. 540–549, IEEE Computer Society, 2004.

[4] Aigner, M. and Ziegler, G. M., *Proofs from THE BOOK*. Springer, 3rd ed., November 2003.

[5] Aldous, D., "The random walk construction of uniform spanning trees and uniform labelled trees," *SIAM J. Discrete Math.*, vol. 3, no. 4, pp. 450–465, 1990.

[6] Aleliunas, R., Karp, R. M., Lipton, R. J., Lovász, L., and Rackoff, C., "Random walks, universal traversal sequences, and the complexity of maze problems," in *FOCS*, pp. 218–223, 1979.

[7] Alon, N., Avin, C., Koucký, M., Kozma, G., Lotker, Z., and Tuttle, M. R., "Many random walks are faster than one," in *SPAA*, pp. 119–128, 2008.

[8] Alon, N., Matias, Y., and Szegedy, M., "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, 1999. Also in STOC'96.

[9] Arge, L. and Vitter, J. S., "Optimal dynamic interval management in external memory," in *FOCS*, pp. 560–569, 1996.

[10] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M., "Proof verification and the hardness of approximation problems," *J. ACM*, vol. 45, no. 3, pp. 501–555, 1998.

[11] Arora, S. and Safra, S., "Probabilistic checking of proofs: A new characterization of NP," *J. ACM*, vol. 45, no. 1, pp. 70–122, 1998. Also appeared in FOCS'92.

[12] Baala, H., Flauzac, O., Gaber, J., Bui, M., and El-Ghazawi, T. A., "A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks," *J. Parallel Distrib. Comput.*, vol. 63, no. 1, pp. 97–104, 2003.

[13] Babai, L., Frankl, P., and Simon, J., "Complexity classes in communication complexity theory," in *FOCS*, pp. 337–347, 1986.

[14] Balke, W.-T., Güntzer, U., and Zheng, J. X., "Efficient distributed skylining for web information systems," in *EDBT*, pp. 256–273, 2004.

[15] Bar-Ilan, J. and Zernik, D., "Random leaders and random spanning trees," in *WDAG (later called DISC)*, pp. 1–12, 1989.

[16] Bar-Yossef, Z., Jayram, T. S., Kumar, R., and Sivakumar, D., "An information statistics approach to data stream and communication complexity," *J. Comput. Syst. Sci.*, vol. 68, no. 4, pp. 702–732, 2004. Also in FOCS'02.

[17] Barndorff-Nielsen, O. and Sobel, M., "On the distribution of the number of admissible points in a vector random sample," *Theory of Probability and its Applications*, vol. 11, no. 2, pp. 249–269, 1966.

[18] Batu, T., Fortnow, L., Fischer, E., Kumar, R., Rubinfeld, R., and White, P., "Testing random variables for independence and identity," in *FOCS*, pp. 442–451, 2001.

[19] Bernard, T., Bui, A., and Flauzac, O., "Random distributed self-stabilizing structures maintenance," in *ISSADS*, pp. 231–240, 2004.

[20] Bharambe, A. R., Agrawal, M., and Seshan, S., "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM*, pp. 353–366, 2004.

[21] Börzsönyi, S., Kossmann, D., and Stocker, K., "The skyline operator," in *ICDE*, pp. 421–430, 2001.

[22] Broder, A. Z., "Generating random spanning trees," in *FOCS*, pp. 442–447, 1989.

[23] Bui, M., Bernard, T., Sohier, D., and Bui, A., "Random walks in distributed computing: A survey," in *IICS*, pp. 1–14, 2004.

[24] Chakrabarti, A., Cormode, G., and McGregor, A., "Robust lower bounds for communication and stream computation," in *STOC*, pp. 641–650, 2008.

[25] Chakrabarti, A., Cormode, G., and McGregor, A., "Annotations in data streams," in *ICALP (1)*, pp. 222–234, 2009.

[26] Chakrabarti, A., Jayram, T. S., and Patrascu, M., "Tight lower bounds for selection in randomly ordered streams," in *SODA*, pp. 720–729, 2008.

[27] Chan, C.-Y., Eng, P.-K., and Tan, K.-L., "Efficient processing of skyline queries with partially-ordered domains," in *ICDE'05*, (Washington, DC, USA), pp. 190–191, IEEE Computer Society, 2005.

[28] Chan, C. Y., Eng, P.-K., and Tan, K.-L., "Stratified computation of skylines with partially-ordered domains," in *SIGMOD Conference*, pp. 203–214, 2005.

[29] Chattopadhyay, A. and Pitassi, T., "The Story of Set Disjointness," *SIGACT News*, vol. 41, no. 3, pp. 59–85, 2010.

[30] Chaudhuri, S., Dalvi, N., and Kaushik, R., "Robust cardinality and cost estimation for skyline operator," in *ICDE'06*, (Washington, DC, USA), p. 64, IEEE Computer Society, 2006.

[31] Chen, L., Cui, B., Xu, L., and Shen, H. T., "Distributed cache indexing for efficient subspace skyline computation in p2p networks," in *DASFAA (1)*, pp. 3–18, 2010.

[32] Chomicki, J., Godfrey, P., Gryz, J., and Liang, D., "Skyline with presorting: Theory and optimizations," in *Intelligent Information Systems*, pp. 595–604, 2005. Also appeared in ICDE'03.

[33] Chu, J. I. and Schnitger, G., "The communication complexity of several problems in matrix computation," *J. Complexity*, vol. 7, no. 4, pp. 395–407, 1991.

[34] Cohen, E., "Size-Estimation Framework with Applications to Transitive Closure and Reachability," *J. Comput. Syst. Sci.*, vol. 55, no. 3, pp. 441–453, 1997. Also in FOCS'94.

[35] Cooper, B. F., "Quickly routing searches without having to move content," in *IPTPS*, pp. 163–172, 2005.

[36] Cooper, C., Frieze, A., and Radzik, T., "Multiple random walks in random regular graphs," in *Preprint*, 2009.

[37] Coppersmith, D., Tetali, P., and Winkler, P., "Collisions among random walks on a graph," *SIAM J. Discret. Math.*, vol. 6, no. 3, pp. 363–374, 1993.

[38] Cormode, G., Mitzenmacher, M., and Thaler, J., "Streaming graph computations with a helpful advisor," *CoRR*, vol. abs/1004.2899, 2010.

[39] Cui, B., Chen, L., Xu, L., Lu, H., Song, G., and Xu, Q., "Efficient skyline computation in structured peer-to-peer systems," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 7, pp. 1059–1072, 2009.

[40] Das Sarma, A., Gollapudi, S., and Panigrahy, R., "Estimating pagerank on graph streams," in *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 69–78, ACM, 2008.

[41] Das Sarma, A., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., and Wattenhofer, R., "Distributed Verification and Hardness of Distributed Approximation," in *STOC*, 2011.

[42] Das Sarma, A., Lall, A., Nanongkai, D., and Xu, J., "Randomized Multi-Pass Streaming Skyline Algorithms," in *VLDB*, pp. 85–96, 2009.

[43] Das Sarma, A., Lipton, R. J., and Nanongkai, D., "Best-Order Streaming Model," in *TAMC*, pp. 178–191, 2009. Invited to Special Issue of Theoretical Computer Science.

[44] Das Sarma, A., Nanongkai, D., and Pandurangan, G., "Fast distributed random walks," in *PODC*, pp. 161–170, 2009.

[45] Das Sarma, A., Nanongkai, D., Pandurangan, G., and Tetali, P., "Efficient distributed random walks with applications," in *PODC*, pp. 201–210, 2010.

[46] Demetrescu, C., Finocchi, I., and Ribichini, A., "Trading off space for passes in graph streaming problems," in *SODA*, pp. 714–723, 2006.

[47] Ding, X. and Jin, H., "Efficient and progressive algorithms for distributed skyline queries over uncertain data," in *ICDCS*, pp. 149–158, 2010.

[48] Dinur, I., "The pcp theorem by gap amplification," *J. ACM*, vol. 54, no. 3, p. 12, 2007. Also appeared in STOC'06.

[49] Dolev, S., Schiller, E., and Welch, J. L., "Random walk for self-stabilizing group communication in ad hoc networks," *IEEE Trans. Mob. Comput.*, vol. 5, no. 7, pp. 893–905, 2006. also in PODC'02.

[50] Dolev, S. and Tzachar, N., "Spanders: distributed spanning expanders," in *SAC*, pp. 1309–1314, 2010.

[51] Dubhashi, D. P., Grandioni, F., and Panconesi, A., "Distributed Algorithms via LP Duality and Randomization," in *Handbook of Approximation Algorithms and Metaheuristics*, Chapman and Hall/CRC, 2007.

[52] Elkin, M., "Distributed approximation: a survey," *SIGACT News*, vol. 35, no. 4, pp. 40–57, 2004.

[53] Elkin, M., "Computing almost shortest paths," *ACM Transactions on Algorithms*, vol. 1, no. 2, pp. 283–323, 2005. Also in PODC'01.

[54] Elkin, M., "A faster distributed protocol for constructing a minimum spanning tree," *J. Comput. Syst. Sci.*, vol. 72, no. 8, pp. 1282–1308, 2006. Also in SODA'04.

[55] Elkin, M., "An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem," *SIAM J. Comput.*, vol. 36, no. 2, pp. 433–456, 2006. Also in STOC'04.

[56] Elsässer, R. and Sauerwald, T., "Tight bounds for the cover time of multiple random walks," in *ICALP (1)*, pp. 415–426, 2009.

[57] Feigenbaum, J., Kannan, S., Strauss, M., and Viswanathan, M., "Testing and spot-checking of data streams," in *SODA*, pp. 165–174, 2000.

[58] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and 0004, J. Z., "Graph distances in the data-stream model," *SIAM J. Comput.*, vol. 38, no. 5, pp. 1709–1727, 2008. Also appeared in SODA'05.

[59] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J., "On graph problems in a semi-streaming model," *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 207–216, 2005.

[60] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S., "Cache-oblivious algorithms," in *FOCS*, pp. 285–298, 1999.

[61] Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L., "Peer-to-peer membership management for gossip-based protocols," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 139–149, 2003.

[62] Garay, J. A., Kutten, S., and Peleg, D., "A Sublinear Time Distributed Algorithm for Minimum-Weight Spanning Trees," *SIAM J. Comput.*, vol. 27, no. 1, pp. 302–316, 1998. Also in FOCS '93.

[63] Gkantsidis, C., Goel, G., Mihail, M., and Saberi, A., "Towards topology aware networks," in *IEEE INFOCOM*, 2007.

[64] Gkantsidis, C., Mihail, M., and Saberi, A., "Hybrid search schemes for unstructured peer-to-peer networks," in *INFOCOM*, pp. 1526–1537, 2005.

[65] Godfrey, P., Shipley, R., and Gryz, J., "Algorithms and analyses for maximal vector computation," *VLDB J.*, vol. 16, no. 1, pp. 5–28, 2007. Also appeared in VLDB'05.

[66] Goldreich, O., *Handbook of massive data sets*, ch. Property testing in massive graphs, pp. 123–147. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[67] Goldwasser, S., Kalai, Y. T., and Rothblum, G. N., "Delegating computation: interactive proofs for muggles," in *STOC*, pp. 113–122, 2008.

[68] Gopalan, P. and Radhakrishnan, J., "Finding duplicates in a data stream," in *SODA*, pp. 402–411, 2009.

[69] Goyal, N., Rademacher, L., and Vempala, S., "Expanders via random spanning trees," in *SODA*, 2009.

[70] Greenwald, M. and Khanna, S., "Space-efficient online computation of quantile summaries," in *SIGMOD '01*, (New York, NY, USA), pp. 58–66, ACM, 2001.

[71] Guha, S. and McGregor, A., "Approximate quantiles and the order of the stream," in *PODS*, pp. 273–279, 2006.

[72] Guha, S. and Mcgregor, A., "Lower bounds for quantile estimation in random-order and multi-pass streaming," in *ICALP*, pp. 704–715, 2007.

[73] Hajnal, A., Maass, W., and Turán, G., "On the communication complexity of graph properties," in *STOC*, pp. 186–191, 1988.

[74] Håstad, J. and Wigderson, A., "The randomized communication complexity of set disjointness," *Theory of Computing*, vol. 3, no. 1, pp. 211–219, 2007.

[75] Hastings, W. K., "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, pp. 97–109, April 1970.

[76] Henzinger, M. R., Raghavan, P., and Rajagopalan, S., "Computing on data streams," in *External memory algorithms* (Abello, J. M. and Vitter, J. S., eds.), pp. 107–118, Boston, MA, USA: American Mathematical Society, 1999.

[77] Israeli, A. and Jalfon, M., "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *PODC*, pp. 119–131, 1990.

[78] Jain, R., Radhakrishnan, J., and Sen, P., "A Direct Sum Theorem in Communication Complexity via Message Compression," in *ICALP*, pp. 300–315, 2003.

[79] Jerrum, M. and Sinclair, A., "Approximating the permanent," *SIAM Journal of Computing*, vol. 18, no. 6, pp. 1149–1178, 1989.

[80] Jiang, B. and Pei, J., "Online interval skyline queries on time series," in *ICDE*, 2009.

[81] Kalyanasundaram, B. and Schnitger, G., "The Probabilistic Communication Complexity of Set Intersection," *SIAM J. Discrete Math.*, vol. 5, no. 4, pp. 545–557, 1992.

[82] Karger, D. R. and Ruhl, M., "Simple efficient load balancing algorithms for peer-to-peer systems," in *SPAA*, pp. 36–43, 2004.

[83] Kelner, J. and Madry, A., "Faster generation of random spanning trees," in *IEEE FOCS*, 2009.

[84] Kempe, D. and McSherry, F., "A decentralized algorithm for spectral analysis," *Journal of Computer and System Sciences*, vol. 74(1), pp. 70–83, 2008.

[85] Kempe, D., Kleinberg, J. M., and Demers, A. J., "Spatial gossip and resource location protocols," in *STOC*, pp. 163–172, 2001.

[86] Khan, M., Kuhn, F., Malkhi, D., Pandurangan, G., and Talwar, K., "Efficient distributed approximation algorithms via probabilistic tree embeddings," in *PODC*, pp. 263–272, 2008.

[87] Khan, M. and Pandurangan, G., "A fast distributed approximation algorithm for minimum spanning trees," *Distributed Computing*, vol. 20, no. 6, pp. 391–402, 2008. Also in DISC'06.

[88] Kleinberg, J. M., "The small-world phenomenon: an algorithmic perspective," in *STOC*, pp. 163–170, 2000.

[89] Knuth, D. E., *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2009.

[90] Kor, L., Korman, A., and Peleg, D., "Tight bounds for distributed mst verification," in *STACS*, pp. 69–80, 2011.

[91] Korman, A. and Kutten, S., "Distributed verification of minimum spanning trees," *Distributed Computing*, vol. 20, no. 4, pp. 253–266, 2007. Also in PODC'06.

[92] Kossmann, D., Ramsak, F., and Rost, S., "Shooting stars in the sky: an online algorithm for skyline queries," in *VLDB'02*, pp. 275–286, VLDB Endowment, 2002.

[93] Kuhn, F., Moscibroda, T., and Wattenhofer, R., "What cannot be computed locally!," in *PODC*, pp. 300–309, 2004.

[94] Kung, H. T., Luccio, F., and Preparata, F. P., "On finding the maxima of a set of vectors," *J. ACM*, vol. 22, no. 4, pp. 469–476, 1975. Also in FOCS'74.

[95] Kushilevitz, E. and Nisan, N., *Communication complexity*. New York, NY, USA: Cambridge University Press, 1997.

[96] Kutten, S. and Peleg, D., "Fast Distributed Construction of Small k-Dominating Sets and Applications," *J. Algorithms*, vol. 28, no. 1, pp. 40–66, 1998. Also in PODC'95.

[97] Lam, T. W. and Ruzzo, W. L., "Results on communication complexity classes," *J. Comput. Syst. Sci.*, vol. 44, no. 2, pp. 324–342, 1992. Also appeared in Structure in Complexity Theory Conference 1989.

[98] Law, C. and Siu, K.-Y., "Distributed construction of random expander networks," in *INFOCOM*, 2003.

[99] Leighton, F. T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1 ed., 1991.

[100] Lengauer, T., "VLSI theory," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pp. 835–868, Elsevier and MIT Press, 1990.

[101] Li, F., Yi, K., Hadjieleftheriou, M., and Kollios, G., "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *VLDB*, pp. 147–158, 2007.

[102] Lin, X., Yuan, Y., Wang, W., and Lu, H., "Stabbing the sky: Efficient skyline computation over sliding windows," in *ICDE*, pp. 502–513, 2005.

[103] Linial, N., "Locality in distributed graph algorithms," *SIAM J. Comput.*, vol. 21, no. 1, pp. 193–201, 1992.

[104] Lipton, R. J., "Fingerprinting sets," cs-tr-212-89, Princton University, 1989.

[105] Lipton, R. J., "Efficient checking of computations," in *STACS*, pp. 207–215, 1990.

[106] Loguinov, D., Kumar, A., Rai, V., and Ganesh, S., "Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience," in *SIGCOMM*, pp. 395–406, 2003.

[107] Lotker, Z., Patt-Shamir, B., and Peleg, D., "Distributed MST for constant diameter graphs," *Distributed Computing*, vol. 18, no. 6, pp. 453–460, 2006. Also in PODC'01.

[108] Luby, M., "A simple parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, vol. 15, no. 4, pp. 1036–1053, 1986. Also in STOC'85.

[109] Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S., "Search and replication in unstructured peer-to-peer networks," in *ICS*, pp. 84–95, 2002.

[110] Lynch, N., *Distributed Algorithms*. Morgan Kaufmann, 1996.

[111] Lyons, R., "Asymptotic enumeration of spanning trees," *Combinatorics, Probability & Computing*, vol. 14, no. 4, pp. 491–522, 2005.

[112] Matousek, J., "Computing dominances in e^n," *Inf. Process. Lett.*, vol. 38, no. 5, pp. 277–278, 1991.

[113] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E., "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[114] Mitzenmacher, M. and Upfal, E., *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. New York, NY, USA: Cambridge University Press, 2005.

[115] Morales, R. and Gupta, I., "Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems," in *ICDCS*, p. 55, 2007.

[116] Morse, M. D., Patel, J. M., and Jagadish, H. V., "Efficient skyline computation over low-cardinality domains," in *VLDB*, pp. 267–278, 2007.

[117] Munro, J. I. and Paterson, M., "Selection and sorting with limited storage," *Theor. Comput. Sci.*, vol. 12, pp. 315–323, 1980. Also appeared in FOCS'78.

[118] Muthukrishnan, S., "Data streams: Algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, vol. 1, no. 2, 2005.

[119] Muthukrishnan, S. and Pandurangan, G., "The bin-covering technique for thresholding random geometric graph properties," in *ACM SODA*, 2005. To appear in Journal of Computer and System Sciences.

[120] Nanongkai, D., Das Sarma, A., and Pandurangan, G., "A Tight Unconditional Lower Bound on Distributed Random Walk Computation," in *PODC*, 2011.

[121] Nisan, N. and Wigderson, A., "Rounds in communication complexity revisited," *SIAM J. Comput.*, vol. 22, no. 1, pp. 211–219, 1993. Also in STOC'91.

[122] Pandurangan, G. and Khan, M., "Theory of communication networks," in *Algorithms and Theory of Computation Handbook, Second Edition*, CRC Press, 2009.

[123] Papadias, D., Tao, Y., Fu, G., and Seeger, B., "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005. Also appeard in SIGMOD'03.

[124] Papadimitriou, C. H. and Sipser, M., "Communication complexity," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 260–269, 1984. Also appeared in STOC'82.

[125] Papadopoulos, S., Yang, Y., and Papadias, D., "Cads: Continuous authentication on data streams," in *VLDB*, pp. 135–146, 2007.

[126] Park, S., Kim, T., Park, J., Kim, J., and Im, H., "Parallel skyline computation on multicore architectures," in *ICDE*, 2009.

[127] Pei, J., Jiang, B., Lin, X., and Yuan, Y., "Probabilistic skylines on uncertain data," in *VLDB*, pp. 15–26, 2007.

[128] Peleg, D., *Distributed computing: a locality-sensitive approach*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[129] Peleg, D. and Rubinovich, V., "A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction," *SIAM J. Comput.*, vol. 30, no. 5, pp. 1427–1442, 2000. Also in FOCS'99.

[130] Raz, R. and Spieker, B., "On the "log rank" – Conjecture in Communication Complexity," in *FOCS*, 1993.

[131] Razborov, A. A., "On the Distributional Complexity of Disjointness," *Theor. Comput. Sci.*, vol. 106, no. 2, pp. 385–390, 1992. Also in ICALP'90.

[132] Ruhl, J. M., *Efficient algorithms for new computational models*. PhD thesis, Massachusetts Institute of Technology, 2003. Supervisor-David R. Karger.

[133] Sacharidis, D., Papadopoulos, S., and Papadias, D., "Topologically sorted skylines for partially ordered domains," in *ICDE*, 2009.

[134] Sami, R. and Twigg, A., "Lower bounds for distributed markov chain problems," *CoRR*, vol. abs/0810.5263, 2008.

[135] Schweikardt, N., "Machine models and lower bounds for query processing," in *PODS*, pp. 41–52, 2007.

[136] Schweikardt, N., "Lower bounds for multi-pass processing of multiple data streams," in *STACS*, pp. 51–61, 2009.

[137] Sipser, M., *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[138] Tan, K.-L., Eng, P.-K., and Ooi, B. C., "Efficient progressive skyline computation," in *VLDB '01*, (San Francisco, CA, USA), pp. 301–310, Morgan Kaufmann Publishers Inc., 2001.

[139] Tao, Y., Ding, L., Lin, X., and Pei, J., "Distance-based representative skyline," in *ICDE*, 2009.

[140] Tao, Y. and Papadias, D., "Maintaining sliding window skylines on data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 2, pp. 377–391, 2006.

[141] Tarjan, R. E., "Applications of path compression on balanced trees," *J. ACM*, vol. 26, no. 4, pp. 690–715, 1979.

[142] Thurimella, R., "Sub-Linear Distributed Algorithms for Sparse Certificates and Biconnected Components," *J. Algorithms*, vol. 23, no. 1, pp. 160–179, 1997. Also in PODC'95.

[143] Torlone, R., Ciaccia, P., and Romatre, U., "Which are my preferred items," in *In Workshop on Recommendation and Personalization in E-Commerce*, pp. 1–9, 2002.

[144] Vazirani, V. V., *Approximation Algorithms*. Springer, July 2001.

[145] Vitter, J. S., "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985. Also appeared in FOCS'83.

[146] Vitter, J. S., "Algorithms and data structures for external memory," *Foundations and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2006.

[147] Vlachou, A., Doulkeridis, C., Kotidis, Y., and Vazirgiannis, M., "Skypeer: Efficient subspace skyline computation over distributed data," in *ICDE*, pp. 416–425, 2007.

[148] Wong, R. C.-W., Fu, A. W.-C., Pei, J., Ho, Y. S., Wong, T., and Liu, Y., "Efficient skyline querying with variable user preferences on nominal attributes," *PVLDB*, vol. 1, no. 1, pp. 1032–1043, 2008.

[149] Wong, R. C.-W., Pei, J., Fu, A. W.-C., and Wang, K., "Online skyline analysis with dynamic preferences on nominal attributes," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 1, pp. 35–49, 2009.

[150] Wu, B. Y., Lancia, G., Bafna, V., Chao, K.-M., Ravi, R., and Tang, C. Y., "A polynomial-time approximation scheme for minimum routing cost spanning trees," *SIAM J. Comput.*, vol. 29, no. 3, pp. 761–778, 1999. Also in SODA'98.

[151] Yao, A. C.-C., "Probabilistic Computations: Toward a Unified Measure of Complexity," in *FOCS*, pp. 222–227, 1977.

[152] Yao, A. C.-C., "Some complexity questions related to distributive computing," in *STOC*, pp. 209–213, 1979.

[153] Yi, K., Li, F., Hadjieleftheriou, M., Kollios, G., and Srivastava, D., "Randomized synopses for query assurance on data streams," in *ICDE*, pp. 416–425, 2008.

[154] Zhang, W., Lin, X., Zhang, Y., Wang, W., and Yu, J. X., "Probabilistic skyline operator over sliding windows," in *ICDE*, 2009.

[155] Zhong, M. and Shen, K., "Random walk based node sampling in self-organizing networks," *Operating Systems Review*, vol. 40, no. 3, pp. 49–55, 2006.

[156] Zhong, M., Shen, K., and Seiferas, J. I., "Non-uniform random membership management in peer-to-peer networks," in *INFOCOM*, pp. 1151–1161, 2005.

[157] Zhu, L., Tao, Y., and Zhou, S., "Distributed skyline retrieval with low bandwidth consumption," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 3, pp. 384–400, 2009.